TU WIEN Informatics

Automated reasoning in linear arithmetic

MASTERARBEIT

zur Erlangung des akademischen Grades

Master of Science

im Rahmen des Studiums

Logic and Computation

eingereicht von

Johannes Felzmann, Bsc
Matrikelnummer 11912368

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof.in Dr.in techn. MSc Laura Kovács

Wien, 6. September 2025

Johannes Felzmann

Laura Kovács

TU WIEN Informatics

Automated reasoning in linear arithmetic

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Logic and Computation

by

Johannes Felzmann, Bsc
Registration Number 11912368

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof.in Dr.in techn. MSc Laura Kovács

Vienna, September 6, 2025

Johannes Felzmann

Laura Kovács

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Johannes Felzmann, Bsc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 6. September 2025

Johannes Felzmann

Danksagung

Ich möchte meine tiefste Dankbarkeit gegenüber Univ.Prof.in Dr.in techn. MSc Laura Kovács für ihre fachkundige Anleitung, kontinuierliche Unterstützung und ihr aufschlussreiches Feedback während der gesamten Dauer dieser Arbeit zum Ausdruck bringen. Sie hat meine Fragen stets sehr schnell beantwortet und sich immer die Zeit genommen, detaillierte Erklärungen zu geben, die mir sehr beim Fortschritt geholfen haben. Ihr konstruktives Feedback und ihre Ermutigung waren maßgeblich für den erfolgreichen Abschluss dieser Arbeit. Besonderer Dank gilt meinen Kolleginnen und Kollegen sowie Freundinnen und Freunden für ihre stetige Motivation und ihre Hilfsbereitschaft. Ebenso möchte ich meiner Familie meinen tiefsten Dank aussprechen, die mich während meines Studiums stets unterstützt und motiviert hat.

Wie Edsger W. Dijkstra einst sagte: „Informatik beschäftigt sich nicht mehr mit Computern als die Astronomie mit Teleskopen.“ Dies erinnert uns daran, dass, obwohl neueste Technologien wichtig sind, das wahre Wesen der Informatik im Verständnis grundlegender Prinzipien und Konzepte liegt. Es ist essenziell, über die neuesten Trends und Innovationen hinauszublicken und sich auf die zugrundeliegenden Ideen und Herausforderungen zu konzentrieren, die den Fortschritt in diesem Fachgebiet vorantreiben.

Acknowledgements

I would like to express my deepest gratitude to Univ.Prof.in Dr.in techn. MSc Laura Kovács for her expert guidance, continuous support, and insightful feedback throughout the course of this thesis. She always answered my questions very quickly and generously took the time to provide detailed explanations, which greatly helped me progress. Her insightful feedback and encouragement were instrumental in the successful completion of this thesis. Special thanks go to my colleagues and friends for their helpful discussions and motivation. I would also like to express my deepest appreciation to my family for being able to support me through these studies and for always motivating me to persevere.

As Edsger W. Dijkstra famously said, „Computer science is no more about computers than astronomy is about telescopes.“ This reminds us that, while technology and tools are important, the true essence of computer science lies in understanding fundamental principles and concepts. It is essential to look beyond the latest trends and innovations, and to focus on the underlying ideas and challenges that drive progress in the field.

Kurzfassung

Automated Reasoning über quantifizierte Formeln mit linearer Arithmetik stellt ein fundamentales Problem in der automatischen Verifikation von Hard- und Software, der Programmsynthese, der Verifikation neuronaler Netze und vielen weiteren Anwendungsbereichen dar. Dennoch bleibt es aufgrund der Quantoren und arithmetischen Theorien eine große Herausforderung. Diese Arbeit bietet einen umfassenden Überblick sowie eine detaillierte Literaturanalyse und vergleicht drei Familien von Ansätzen: superpositionsbasierte Kalküle, instantiierungsbasierte Frameworks und algebraische Methoden der Quantoreneliminierung. Die Ergebnisse zeigen, dass superpositionsbasierte Verfahren insbesondere bei Problemen, die Arithmetik mit uninterpretierten Funktionen kombinieren, sehr effektiv sind, während Instantiierungsmethoden bei wenigen Quantorenalternationen besonders stark abschneiden. Approximative Quantoreneliminierung hingegen erreicht eine gute Skalierbarkeit, allerdings auf Kosten der Vollständigkeit. Diese Erkenntnisse verdeutlichen, dass ein einzelner Ansatz nicht alle Problemklassen dominiert, weshalb hybride Architekturen den vielversprechendsten Weg für zukünftige Entwicklungen darstellen.

Abstract

Automated Reasoning involving quantified formulas over linear arithmetic is a fundamental problem in automated verification of hardware and software, program synthesis, neural network verification and many more other areas. Nevertheless it remains highly challenging due to the interaction of quantifiers and arithmetic theories. This thesis provides a comprehensive and detailed study with a thorough literature search and comparison between three families of approaches: superposition-based calculi, instantiation-based frameworks, and algebraic quantifier elimination methods. The results show that superposition-based lifting is particularly effective on problems involving arithmetic with uninterpreted functions, instantiation methods perform strongly on few quantifier alternations, and approximative quantifier elimination achieves good scalability at the cost of completeness. These findings indicate that no single approach is outstanding across all problem classes, and suggests implying that hybrid architectures offer the most promising path forward.

Contents

CHAPTER 1

Introduction

Automated reasoning plays a central role in the verification, synthesis and analysis of computational systems. There exist several logical theories that underlie this field, but linear arithmetic (LA) stands out as one of the most frequently used. While LA is really important, at the same time it is also one of the most challenging theories in the presence of quantifiers and uninterpreted function symbols. In this context, LA involves mostly Linear Real Arithmetic (LRA) and Linear Integer Arithmetic (LIA). Its expressive power, combined with its numerical expressiveness in software and hardware systems, makes it a foundation for modern Satisfiability Modulo Theories (SMT) solving and theorem proving.

Automated reasoning in LA is used across many applications in computer science, engineering and applied mathematics. The most important applications are formal verification of hardware and software, program synthesis, SMT solving, theorem proving, cryptographic protocol verification and neural network verification [Cook, 2018], [Desharnais et al., 2022], [Distefano et al., 2019], [Armando and Compagna, 2004], [Ehlers, 2017], [Sidrane et al., 2021], [So et al., 2019], [Nehai and Bobot, 2019]

While the quantifier-free fragment of LA is well understood and can be efficiently solved with modern SMT solvers, the introduction of quantifiers remains a big challenge. But why do we even need quantified LA? It enables us to express much deeper and more general properties. In general, quantifier-free formulas answer questions like „Is there a solution?“, but quantified formulas address much more deeper questions like „Does a solution exist for all possible scenarios?“ or „Can we find a solution that works for some critical case?“.

Let us have a look at an example, why quantified LA is needed for practical applications. Consider some blockchain platform (e.g. Ethereum), where smart contracts are used to

manage token transfers, enforce financial constraints, and guarantee security properties. Imagine a smart contract that implements a token transfer function. The function can be modeled with the following constraints:

- Before transfer:
  - $balance_{from} \geq amount$
- After transfer:
  - $balance'_{from} = balance_{from} - amount$
  - $balance'_{to} = balance_{to} + amount$

It is easy to observe what the constraints are stating. The balance of the user who wants to do the transfer needs to be bigger or at least equal to the amount of the transfer beforehand, and after the balance is reduced, the balance of the receiving user has the new amount added. If we now want to automatically verify that no user ends up with a negative balance, we could express this with the following formula: $\forall from, amount(balance_{from} \geq amount) \implies (balance_{from} - amount \geq 0)$.

We can observe that the quantifier in the formula is really important, because with it we can formally prove that for all users and transfer amounts, the contract maintains safety. Without quantifiers, we need to test specific values, which would lead to an explosion in formulas.

As we have seen, automated reasoning with quantified formulas in LRA and LIA is really important in practice, but it is also really difficult. Although both theories are decidable, existing quantifier elimination algorithms are often expensive due to their worst-case exponential or even double- or triple-exponential complexity. And even further, real-world quantified formulas frequently mix uninterpreted functions, arrays, or non-linear constructs, resulting in even more complexity.

There exist a lot of methods to handle quantifiers with LA. This thesis presents a comprehensive survey of approaches for automated reasoning in quantified LA, organized according to the following three methodological pillars:

- Superposition-Based Methods
  - These methods extend resolution by applying inference rules to equalities and other logical clauses to derive conclusions or prove unsatisfiability. See [Korovin et al., 2023], [Baumgartner et al., 2015] or [Althaus et al., 2009].
- Instantiation-Based Methods:

  - These methods solve logical formulas by replacing quantified variables with specific terms or values, turning general statements into simpler, concrete cases that can be checked more easily. See [Reynolds et al., 2017], [Ganzinger and Korovin, 2006] or [Niemetz et al., 2021].

- Quantifier Elimination Methods in Computer Algebra:
  - These methods remove quantifiers from logical formulas by transforming them into equivalent quantifier-free expressions, enabling easier analysis and decision procedures. See [Schoisswohl et al., 2024], [Garcia-Contreras et al., 2023] or [Bjørner and Janota, 2015].

The thesis analyzes the complexity, soundness and completeness, practical use cases, experimental results and limitations and future work of each approach. *The goal is to provide both a systematic overview and a critical comparison.* Through this survey, the thesis aims to clarify the current landscape of quantified reasoning in linear arithmetic and to identify promising directions for future research and integration into automated reasoning tools.

Furthermore, one could also mention Automata-Based Methods [Habermehl et al., 2024], [Boigelot et al., 2005], [Boigelot and Wolper, 2002], [Boigelot et al., 2001], [Boigelot et al., 1998], where arithmetic constraints are encoded as automata and the numbers themselves are represented as bitvectors, and then use automata-theoretic operations to reason about quantified formulas. It is also important to mention that there also exist Hybrid Methods like [Nalbach et al., 2023], [Bromberger et al., 2020] or [Ge and de Moura, 2009] that combine various approaches.

CHAPTER 2

Automated Reasoning

2.1 Overview

The introduction section for automated reasoning is following the information from [Portoraro, 2001]. Reasoning in theoretical computer science is defined as the ability to make inferences. Inferences are all the steps which are required in order to move from premises to logical consequences. A premise is a declarative statement that is true or false in an argument. From premises, a logical consequence or also called entailment can be drawn, so it describes the relationship between statements, and one statement follows from one or more statements if they hold true. One of the most famous examples which illustrate the definition of inferences is the following:

1. All humans are mortal.
2. All Greeks are humans.
3. All Greeks are mortal.

Other than checking that the premises and conclusion are true, inference asks if the conclusion follows from the truth of the premises. In the context of this thesis and mostly in this context in general, reasoning is identified with valid deductive reasoning, where valid inferences are drawn. If the conclusion follows logically from its premises, the inference is valid. So it is impossible for the premises to be true and the conclusion to be false. This is also illustrated in the example above.

Automated reasoning can be defined as building computing systems that automate the process of reasoning. With the definition of inference given above, automated reasoning in that sense is related to mechanical theorem proving. Automating the process of reasoning means providing an algorithmic description to a formal calculus so that it can

be implemented on a computer to prove theorems of the calculus in an effective manner. While designing such an algorithmic approach a few steps need to be done. First of all, the class of problem which should be solved needs to be defined and also the language which is used to represent information. After that, the mechanism which the approach will use for deductive inferences need to be chosen and in the last step, how all of these computations are performed needs to be defined.

An automated reasoning program's main purpose is to solve problems. These problems consist of two items: the problem's assumptions and the problem's conclusion. The assumptions are a collection of various statements, which express all the information available to a program, one could also think about it as a knowledge base. The problem's conclusion is a statement indicating the questions asked to an automated reasoning program. To get a better understanding of an automated reasoning program consider the following example:

Problem conclusion: *Do cats live on land?*
Problem assumptions: *Cats are mammals.* and *Mammals live on land.*

The program uses logical deduction and evaluates that the conclusion is true. In this example, cats are mammals and mammals live on land and therefore cats live on land. But note that automated reasoning can not make predictions or generalizations. Consider the following example:

Problem conclusion: *Do all mammals live on land?*
Problem assumptions: *Cats are mammals.* and *Cats live on land.*

In that case, human guidance is needed when performing the deductive reasoning task.

The problem domain of an automated reasoning program can be very large, as for example for a general-purpose theorem prover for first-order logic. However, the domain could also be a more specialised case, such as the modal logic K [Kripke, 1963]. Furthermore, how problems are presented to the reasoning program, represented internally, and how solutions are found depend on the problem domain and the underlying deduction calculus. Mostly first-order logic (FOL), typed λ-calculus [Barendregt et al., 2013] or causal logic [Pearl et al., 2000] is used as a formalism. In the context of this thesis, the focus is on FOL, therefore the reader should be familiar with the basics of FOL.

For a program to be able to perform inferences, a deduction calculus needs to be selected. This choice is also dependent on the problem domain. A deduction calculus basically consists of a set of logical axioms and a set of deduction rules to derive new formulas from previously derived formulas. Putting it all together and trying to solve a problem in the problem domain means, in an abstracted way, to establish a formula α (problem's conclusion) from the set Γ (problem's assumptions), which consists of the

logical and domain axioms and all assumptions. In particular, the automated reasoning program needs to determine if Γ entails α, which is denoted as $\Gamma \vDash \alpha$. How the program internally establishes this semantic fact depends on the implemented calculus. Some programs do that by constructing a step-by-step proof of α from Γ, others try to do that by showing that $\Gamma \cup \{\neg\alpha\}$ is inconsistent, which is done by deriving a contradiction from the set $\Gamma \cup \{\neg\alpha\}$. The former approach usually includes natural deduction systems [Gentzen, 1935], [Prawitz, 2006], while the latter is based on resolution [Robinson, 1965b], sequent deduction [Baumgartner et al., 1996] or matrix connection methods [Bibel, 1983]. A general overview can be found in [Robinson and Voronkov, 2001].

When talking about automated deduction, two properties should be considered: soundness and completeness. Soundness is defined as: all rules of the calculus are truth preserving. More formally, for a direct calculus, it states that if $\Gamma \vdash \alpha$ then $\Gamma \vDash \alpha$ and for an indirect calculus, if $\Gamma \cup \{\neg\alpha\} \vdash \bot$ then $\Gamma \vDash \alpha$. Furthermore, completeness is defined for direct calculi as: if $\Gamma \vDash \alpha$ then $\Gamma \vdash \alpha$. Loosely speaking, it could be seen as the converse of soundness. For an indirect calculus, it is defined in terms of refutations, which means $\Gamma \vDash \alpha$ implies $\Gamma \cup \{\neg\alpha\} \vdash \bot$. Note that soundness is the most desirable property. If a calculus is incomplete, it means that not all entailment relations can be established. In the context of automated reasoning, it is the case that there are satisfiable statements that the program cannot prove. Incompleteness is not that problematic, but lack of soundness is, an unsound reasoning program would be able to generate false conclusions from true assumptions. Also note that there is a difference between a logical calculus and its implementation in an automated reasoning program. The calculi is mostly modified, which results in a new calculus. That is, mostly a mechanization of its deduction rules, which means the specification of the way in which the rules are applied is modified. By doing so, it is important to preserve the metatheoretical properties of the original calculus.

Besides soundness and completeness also decidability and complexity is important for automated deduction. If there exists an algorithm that, for any given Γ and α, can determine in a finite amount of time the answer: „Yes“ or „No“, to the question: „Does $\Gamma \vDash \alpha$?“. Of course, there also exist undecidable calculi, in that case, it needs to be determined which decidable fragment is implemented. Complexity of a calculus talks about time and space, which means how efficient its algorithmic implementation is. Many calculi are not decidable and have really poor complexity measures, which forces, in the context of automated reasoning, researchers to seek trade-offs between the efficiency of an algorithm and deductive power.

Automated reasoning is used for a variety of problems. For example SAT-solving [Biere et al., 2009], logic programming [Lloyd, 2012], mathematics or formal verification for software and hardware [Robinson and Voronkov, 2001], [Grumberg et al., 1999], [Huth and Ryan, 2004].

2.2 Deduction Calculi for Automated Reasoning

Deduction calculi can be viewed as formal systems that define rules and procedures for deriving conclusions from premises in some logical framework. In the context of automated reasoning, they provide the foundation that enables automated reasoning programs to perform sound logical inferences.

Before introducing some basic deduction calculi, we need to review some basics about logical formulas. A literal is the most basic building block, which is either an atom or the negation of an atom. A clause is a logical formula, which is formed by combining a finite set of literals using the logical "or" operator (disjunction). A clause is therefore true if at least one of its literals is true.

2.2.1 Resolution

Robinson pioneered saturation-based theorem proving in its modern form [Robinson, 1965b] as he introduced the resolution calculus. A detailed introduction and some more information can be found in [Bachmair et al., 2001]. Resolution based calculi are the most popular ones implemented in reasoning programs. The chain rule is the main component of resolution, which is a special case of Modus Ponens. In essence, the rule states that from $p \vee q$ and $\neg q \vee r$ one can infer $p \vee r$. To state the rule of ground resolution more formally, assume that C_1 and C_2 are ground clauses. Let $C_1 - l_1$ denote the clause C_1 with the literal l_1 removed. Assume a positive literal l_1 and a negative literal $\neg l_2$, such that l_1 and $\neg l_2$ are complementary. Then the rule states that resolving C_1 and C_2 gives the result $C_1 - l_1 \vee C_2 - \neg l_2$. A significant result for automated deduction is Herbrand's theorem [Herbrand, 1930], which basically assures that ground resolution can establish the non-satisfiablity of any set of clauses, whether they are ground or not. That is important for automated deduction, because it tells us that it can be determined in finitely many steps, whether a set Γ is not satisfied by any of the infinitely many interpretations. This is a great result, but a direct implementation of ground resolution using Herbrand's theorem is very inefficient, since it requires the generation of a large number of ground terms. This problem was addressed by introducing the notion of unification, where the ground resolution rule is generalized to binary resolution. With unification, the instantiation of clauses happens at the moment when they are resolved. Also, the resulting clauses may still contain variables. Note that binary resolution and unification are two of the most important building blocks in the field of automated reasoning.

A unifier of two expressions - terms or clauses - is a substitution that, when applied to the expressions, makes them equal.

Example 2.2.1. The substitution σ given by $\sigma := \{x \leftarrow b, y \leftarrow b, z \leftarrow f(a,b)\}$ is a unifier for $R(x, f(a,y))$ and $R(b,z)$. When applied to both, the substitution makes them equal: $\sigma(R(x, f(a,y))) = R(b, f(a,b)) = \sigma(R(b,z))$

Most general unifier (mgu) produces the most general instance shared by two unifiable expressions.

The substitution in Example 2.3.1 is a unifier, but not a mgu. However the substitution $\sigma' := \{x \leftarrow b, z \leftarrow f(a, y)\}$ is. The process of unification is a central component of most automated deduction programs.

Let C_1 and C_2 be two clauses containing, respectively, a positive literal l_1 and a negative literal $\neg l_2$ such that l_1 and l_2 unify with mgu θ.

$$\frac{C_1 \quad C_2}{(C_1\theta - l_1\theta) \vee (C_2\theta - \neg l_2\theta)} \tag{2.1}$$

$(C_1\theta - l_1\theta) \vee (C_2\theta - \neg l_2\theta)$ is called a binary resolvent of C_1 and C_2.

If two or more literals occurring in a clause C share an mgu θ then $C\theta$ is a factor of C.

Example 2.2.2. Consider $R(x, a) \vee \neg K(f(x), b) \vee R(c, y)$, then with mgu $\{x \leftarrow c, y \leftarrow a\}$ the literals $R(x, a)$ and $R(c, y)$ unify and $R(c, a) \vee \neg K(f(c), b)$ is a factor.

Let C_1 and C_2 be two clauses. Then, a resolvent obtained by resolution from C_1 and C_2 is defined as:

1. a binary resolvent of C_1 and C_2, or
2. a binary resolvent of C_1 and a factor of C_2, or
3. a binary resolvent of a factor of C_1 and C_2, or
4. a binary resolvent of a factor of C_1 and a factor of C_2

A resolution proof, or to be exact refutation, is created by deriving the empty clause [ ] from $\Gamma \cup \{\neg\alpha\}$ using the resolution principle. If $\Gamma \cup \{\neg\alpha\}$ is unsatisfiable, this is always possible, since resolution is refutation complete [Robinson, 1965b].

Example 2.2.3. Show that the set $\Gamma := \{\forall x(P(x) \vee Q(x)), \forall x(P(x) \implies R(x)), \forall x(Q(x) \implies R(x))\}$ entails $\exists x R(x)$.

S_0	1	$P(x) \vee Q(x)$	Assumption
	2	$\neg P(x) \vee R(x)$	Assumption
	3	$\neg P(x) \vee R(x)$	Assumption
	4	$\neg R(a)$	Negate conclusion
S_1	5	$Q(x) \vee R(x)$	Res 1 2
	6	$P(x) \vee R(x)$	Res 1 3
	7	$\neg P(a)$	Res 2 4
	8	$\neg Q(a)$	Res 3 4
S_2	9	$Q(a)$	Res 1 7
	10	$P(a)$	Res 1 8
	11	$R(x)$	Res 2 6
	12	$R(a)$	Res 3 5
	13	$Q(a)$	Res 4 5
	14	$P(a)$	Res 4 6
	15	$R(a)$	Res 5 8
	16	$R(a)$	Res 6 7
S_3	17	$R(a)$	Res 2 10
	18	$R(a)$	Res 2 14
	19	$R(a)$	Res 3 9
	20	$R(a)$	Res 3 13
	21	[ ]	Res 4 11

The resolution proof is deriving [ ] successfully, but there are some drawbacks. The 21 steps are really long, because the proof uses a naive brute-force implementation. Some formulas are redundant, as $R(a)$ is derived six times here, a is a skolem constant. The deduction process of resolution must be supplemented by strategies in order to improve the efficiency. There exist a few resolution strategies. For an efficient deduction calculus implementation in an automated reasoning program, search strategies that reduce the search space are required. Furthermore, there are strategies which remove redundant clauses or tautologies. Robinson introduced a process known as subsumption [Robinson, 1965b], where more specific clauses are removed in the presence of more general ones. There are also strategies which prevent the generation of useless clauses instead of removing redundant clauses. One of the most powerful strategies of this kind is the set-of-support strategy [Wos et al., 1965]. There is also a strategy called hyperresolution [Robinson, 1965a], which reduces the number of intermediate resolvents by combining several resolution steps into a single inference step. Linear resolution [Loveland, 1970] resolves a clause always against the resolvent which was recently derived. The deduction then gets a linear structure for a straightforward implementation, and linear resolution preserves refutation completeness. With linear resolution, the proof in Example 2.3.3 is significantly shortened.

Example 2.2.4. Here, the empty clause is derived from clauses of Example 2.2.3 in eight steps instead of 21.

1	$P(x) \vee Q(x)$	Assumption
2	$\neg P(x) \vee R(x)$	Assumption
3	$\neg P(x) \vee R(x)$	Assumption
4	$\neg R(a)$	Negate conclusion
5	$\neg P(a)$	Res 2 4
6	$Q(a)$	Res 1 5
7	$R(a)$	Res 3 6
8	[ ]	Res 4 7

All strategies mentioned, except unrestricted subsumption, preserve refutation completeness. As discussed before, efficiency is really crucial in automated reasoning, therefore sometimes completeness is traded for speed. Two prominent refinements of linear resolution are unit resolution and input resolution. In unit resolution, one of the resolved clauses is always a literal. In input resolution, one of the resolved clauses is always selected, to be refuted, form the original set. Both strategies are more efficient than standard linear resolution, but neither of them is complete. There also exists an ordered resolution, where clauses are not treated as sets of literals, but as sequences of literals. This strategy is also not refutation complete, but very efficient.

Summing up, we can conclude that it is only possible to enhance certain aspects of the deduction process, at the expense of others. This could be, for example, reducing the size of the proof search space, but on the other hand, increasing the length of the shortest refutations. A more detailed look at theorem-proving strategies can be found in [Bonacina, 2001], complexity results can be found in [Buresh-Oppenheim and Pitassi, 2007] and [Urquhart, 1987].

A very prominent automated reasoning program, based on resolution, is Vampire [Kovács and Voronkov, 2013]. There are also other programs like Otter (Prover4) [Wos et al., 1986].

2.2.2 Term Rewriting

One really fundamental logical relation within automated deduction is equality. Equality logic, and more generally, term rewriting treat equality-like equations as rewrite rules. These rules are also known as reduction or demodulation rules. Considering an equality statement like $f(c) = c$ allows the simplification of a term like $g(b, f(c))$ to $g(b, c)$, but note that the same equation also has the potential to generate an unboundedly large terms: $g(b, f(c)), g(b, f(f(c))), g(b, f(f(f(c)))), \ldots$. There is a difference between equality logic and term rewriting, as the latter equations are used as unidirectional reduction rules as opposed to equality, which works in both directions. Generally, rewrite rules have the form $t_1 \Rightarrow t_2$ and the idea is to look for terms t occurring in expressions e such that t unifies with t_1 with unifier θ, such that the occurrence $t_1\theta$ in $e\theta$ can be replaced by $t_2\theta$.

Example 2.2.5. The rewrite rule $x + 0 \Rightarrow x$ allows the rewriting of $succ(succ(0) + 0)$ to $succ(succ(0))$.

The main ideas of term rewriting can be introduced with an example involving symbolic differentiation, as in Chapter 1 of [Baader and Nipkow, 1998]. Let der denote the derivative with respect to x, let y be a variable different from x and let u and v be variables ranging over expressions. Now it is possible to define the rewrite system:

$$\begin{array}{ll} \text{(R1)} & der(x) \Rightarrow 1 \\ \text{(R2)} & der(y) \Rightarrow 0 \\ \text{(R3)} & der(u+v) \Rightarrow der(u) + der(v) \\ \text{(R4)} & der(u \times v) \Rightarrow (u \times der(v)) + (der(u) \times v) \end{array}$$

This derivation system works as follows. Let's consider the computation of the derivative of $x \times x$ respect to $x, der(x \times x)$:

$$\begin{array}{rl} der(x \times x) \Rightarrow (x \times der(x)) + (der(x) \times x) & \text{by R4} \\ \Rightarrow (x \times 1) + (der(x) \times x) & \text{by R1} \\ \Rightarrow (x \times 1) + (1 \times x) & \text{by R1} \end{array}$$

As none of the rules (R1)-(R4) applies anymore, no further reduction is possible and the rewriting process ends. The final expression is called a normal form. Now, an interesting question arises: Is there an expression whose reduction process will never terminate when applying the rules? Or: Under what conditions a set of rewrite rules will always stop, for any given expression, at a normal form after finitely many applications of the rules? This question is also known as the termination problem of a rewrite system.

It is also possible that when reducing an expression, the set of rules of a rewriting system could be applied in more than one way. This is also the case in the presented derivation system. In the reduction $der(x \times x)$, it is also possible to apply R1 first to the second sub-expression in $(x \times der(x)) + (der(x) \times x)$. If it is done that way, the reduction also terminates with the same normal form as in the previous case. But note that this should not be taken for granted. A rewriting system is said to be (globally) confluent if and only if, independently of the order in which the rules are applied, every expression always ends up in the same normal form. The presented system can be shown to be confluent.

It is also possible to add more rules to a system in order to make it more practical, but note that it can also have some consequences. Adding for example a new rule to the system results in losing confluency:

$$\text{(R5)} \quad u + 0 \Rightarrow u$$

Now it is possible to show that $der(x+0)$ has two normal forms:

$$\begin{array}{rl} der(x+0) \Rightarrow der(x) + der(0) & \text{by R3} \\ \Rightarrow 1 + der(0) & \text{by R1} \end{array}$$

$$\begin{aligned} der(x+0) &\Rightarrow der(x) && \text{by R5} \\ &\Rightarrow 1 && \text{by R1} \end{aligned}$$

Further, it is possible to add another rule to allow further reduction of $1 + der(0)$:

$$\text{(R6)} \quad der(0) \Rightarrow 0$$

And now, magically, confluence is restored. This raises another question: How can we make a non-confluent system into an equivalent confluent one? There is an answer to this question in [Knuth and Bendix, 1983], namely the Knuth-Bendix completion algorithm.

Of course, term rewriting needs strategies to direct its application, and there are a lot of them. For the scope of this thesis, the most important one is the superposition calculus, as this is used in the superposition-based approaches for quantified reasoning in linear arithmetic. The superposition calculus is a calculus for reasoning in equational first-order logic. The calculus has its origins in the early 1990s and was introduced in [Bachmair and Ganzinger, 1994]. It combines notions from first-order resolution and Knuth-Bendix ordering equality. Furthermore, it is also shown to be refutation complete and is used in many theorem provers, most notably in Vampire. The calculus tries to show the unsatisfiability of a set of first-order logic clauses. Due to its refutation completeness, from any unsatisfiable clause set, a contradiction will eventually be derived given unlimited resources and a derivation strategy that is fair.

2.2.3 Other deduction methods

There also exist other deduction methods like sequent deduction, natural deduction, matrix connection method or mathematical induction. Further information for all these methods can be found in [Portoraro, 2001].

2.3 Applications

2.3.1 Boolean Satisfiability Problem (SAT)

Determining satisfiability of logic formulas is one of the famous problems of computer science, therefore, and also because of great interest from the industry, the automated reasoning community has put a lot of attention on this. As a reminder, a formula is satisfiable if there is an assignment of truth-values to its variables that makes the formula true. For example consider the formula $(P \vee R) \implies Q$ where $(P \leftarrow \text{true}, R \leftarrow \text{true}, Q \leftarrow \text{false})$ does not make the formula true, but $(P \leftarrow \text{false}, R \leftarrow \text{true}, Q \leftarrow \text{true})$ does. The process of determining whether a formula is satisfiable or not with any assignment is called the Boolean Satisfiability Problem (SAT). For a formula with n variables to determine a satisfiability assignment, one has to inspect each of the 2^n possible assignments. This method is complete as: if the formula is satisfiable, then we will eventually find an assignment and also, if the formula is not satisfiable, we will also additionally be able to show

this. But, as promising as this sounds, the search for such an assignment, in particular in the non-satisfiable case, takes an exponential amount of time. A lot of problems such as graph-theoretic problems, network design, scheduling or program optimization can be expressed as SAT-instances, therefore, there is a big desire for more efficient algorithms. SAT is NP-complete [Cook, 1971], so it is very unlikely that a polynomial algorithm will be found, but for sure, there are efficient algorithms for some cases of SAT problems.

One of the first SAT search algorithms was the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [Davis and Putnam, 1960]. This algorithm is still considered to be one of the best complete SAT solvers. DPLL algorithms are mostly extended by some strategies in order to make them more efficient. These strategies are, for example, term indexing [Graf and Fehrer, 1998], where the formula variables are ordered in an advantageous way, chronological backtracking [Nadel and Ryvchin, 2018], where one backtracks to a previous branching point if the process leads to a conflicting clause, or most prominent conflict-driven learning [Lintao et al., 2001], which determines the information to keep and where to backtrack. Of course, all these strategies and even more can be combined, resulting in an even more efficient solver.

2.3.2 Satisfiability modulo theories (SMT)

Satisfiability Modulo Theory (SMT) is an approach of great interest in solving SAT problems in FOL. As SMT is an important topic for this thesis, we will take some time and introduce SMT carefully, mostly following [de Moura and Bjørner, 2009], as this introduction is readable and covers the most important topics. SMT is about checking satisfiability of logical formulas over one or more background theories. The problem of Boolean satisfiability is here combined with domains, and it also draws on the biggest problems in the past century of symbolic logic, which are the decision problem, completeness and incompleteness of theories and also complexity theory. Most SMT-problems have a very high computational complexity. The challenge of integrating specialized algorithms for different domains is just as complex and fascinating as developing new algorithms that perform exceptionally well within such combinations. One prominent theory, which we will introduce later on, is the theory of LA.

SAT is NP-complete, and it is well known that FOL is undecidable. It is really infeasible to come up with a procedure that can solve arbitrary SMT problems due to their high computational complexity. Because of that, most work is focused on the more realistic goal of solving problems that occur in practice efficiently. In the modern era of theoretical computer science, there has been enormous progress in the scale of problems that can be solved. This is due to innovations in algorithms, heuristics and data structures. With modern, efficient SAT procedures, formulas with hundreds of thousands of variables and millions of clauses can be checked. And also, SMT procedures had a similar progress for the more commonly occurring theories. There is an annual competition for SAT and SMT procedures, with the most recent one, "The 20th International Satisfiability Modulo Theories Competition (SMT-COMP 2025)", where the most efficient procedures

are ranked. That competition is also a driving force in the development of more efficient procedures.

In the core, most automated deduction tools use case-analysis. SMT solvers rely mostly on SAT procedures for doing so. Firstly, we analyze the basic techniques of state-of-the-art SAT solvers and later how SMT specific solvers are combined with SAT solvers. Most efficient SAT solvers are based on systematic search. The search space is built like a tree, where each vertex represents a propositional variable and the outgoing edges represent the two choices, true and false, for this variable. This results, for a formula containing n variables, in 2^n leaves in the tree. Every path from the root to a leaf is a truth assignment. For checking truth assignments, a procedure is needed for a formula φ, that is based on systematic search. This procedure searches the tree for a truth assignment M that satisfies φ. The majority of SAT solvers are based on the previously mentioned DPLL approach. This algorithm takes a CNF formula and tries to build a satisfying truth assignment using three operations: decide, propagate and backtrack. Decide chooses an unassigned propositional variable using a heuristic and assigns it to true or false. This is the so called case-splitting. Furthermore, propagate tries to deduce consequences using deduction rules. If the algorithm comes up with a conflict, which indicates that some earlier decisions cannot lead to a truth assignment, DPPL uses backtrack and tries a different branch. If a conflict comes up and there is no decision to backtrack, then the formula is unsatisfiable. There exist a lot of improvements to this procedure to enhance efficiency.

SMT considers background theories. A theory can be defined as a set of sentences. Formally, we can define a Σ-theory as a collection of sentences over a signature Σ. A formula φ, considering a theory T, is satisfiable modulo T if $T \cup \{\varphi\}$ is satisfiable. We use $M \vDash_T \varphi$ instead of $M \vDash \{\varphi\} \cup T$. Consider the following example, where Σ contains the symbols 0, 1, +, - and $<$, and the structure $\mathbb{Z}$ interprets these symbols in the usual way over the integers. The theory of LA is then the set of first-order sentences that are true in $\mathbb{Z}$. Denote Ω as the class of structures over a signature Σ, then we use $Th(\Omega)$, to denote the set of all sentences ϕ over Σ, where $M \vDash \phi$ for every M in Ω. The satisfiability problem for a theory T is decidable if there exists a procedure that checks whether any quantifier-free formula is satisfiable or not. In this case, the procedure is a decision procedure for T.

There are a lot of theories that are integrated with SMT solvers. For example LA, difference arithmetic, non-linear arithmetic, free functions, bit vectors, arrays and many others. In this thesis, the focus is set on LA.

2.3.3 SMT solvers

Back in the days, the most used approach for SMT solvers was to translate SMT instances into SAT instances and pass them to a SAT solver. This approach is called eager approach or bitblasting. For example see [Jha et al., 2009]. The approach has some drawbacks, as

the high-level semantics of the theories are lost. These led to various other approaches for SMT solvers.

SMT solvers determine the satisfiability of FOL formulas extended with background theories. Many automated reasoning scenarios use SMT solvers for solving quantifier-free formulas. These may be sufficient for most applications, but there also exist many use case scenarios where complex first-order quantification is needed. Examples of such scenarios would include expressing arithmetic operations over memory allocation and financial transactions, as discussed in [Alt et al., 2022], [Elad et al., 2021], [Gurfinkel, 2022] and [Passmore, 2021]. Quantifiers are handled in SMT solvers using heuristic instantiation in domain-specific model construction. That approaches can be observed in [Bonacina, 2001], [de Moura and Bjørner, 2007], [De Moura and Jovanović, 2013] and [Reynolds et al., 2017]. Instantiation is incomplete in most cases and requires instances to be produced to perform reasoning. For quantifier-heavy problems, this can lead to an explosion in work. Rather than that, for the above use cases, we need a reasoning approach, which is able to handle both theories and complex applications of quantifiers. A major challenge in both SMT and first-order theorem proving is combining quantifiers with theories and doing so especially with arithmetic.

2.3.4 Theorem provers

Theorem provers are tools designed to establish the validity of logical statements by constructing formal proofs. That means they can determine whether a given logical statement/theorem is true based on a set of inference rules and axioms. There exist automated and also interactive theorem provers. Automated theorem provers attempt to find proofs with no human guidance or at least minimal. For these kinds of provers, many algorithms such as resolution or rewriting are used. Interactive theorem provers, on the other hand, require human guidance to construct proofs.

While theorem proving and SMT solving are closely related, they differ in their scope and techniques. A SMT solver wants to determine if a logical formula is satisfiable given a background theory. A theorem prover intends to prove that a statement follows logically from a set of axioms. Therefore, a SMT solver returns if a formula is satisfiable, unsatisfiable or even unknown, a theorem prover on the other hand, returns a proof or counterexample. SMT solving can be very efficient for specific theories, and theorem proving is more general. Note that a combination of both or even using SMT inside theorem provers or using theorem proving inside SMT solvers is possible and very powerful, as they complement each other's strengths. For example see [Voronkov, 2014].

CHAPTER 3

Linear Arithmetic

3.1 Overview

Linear arithmetic (LA) focuses on relationships expressed through linear equations and inequalities. Considering the relationships, variables appear with a maximum degree of one and are combined using operations like addition, subtraction, and multiplication by constants.

Basically, LA deals with equations and inequalities of the form $a_1x_1+a_2x_2+\cdots+a_nx_n \sim b$, where $x_1, x_2, \ldots, x_n$ are variables, $a_1, a_2, \ldots, a_n, b$ are constants and $\sim$ represents relational operators like $=, >, <, \geq, \leq$. As the name suggests, a key feature of LA is that the relationships are strictly linear. This essentially means that variables are not multiplied together or raised to powers greater than one. To make that clearer, consider the following example.

Example 3.1.1. $3x + 2y = 6$ is a linear equation, while $x^2 + y = 5$ is a non-linear equation due to the term x^2.

The most prominent categoreis of LA are:

- **Linear integer arithmetic (LIA) (aka. Presburger Arithmetic)**, where variables are restricted to integer values.
- **Linear real arithmetic (LRA)**, where variables can take any real value.
- **Linear integer-real arithmetic (LIRA)**, which combines both integer and real variables.

It is important to define which category is considered, as each category has its unique challenges and applications, depending on whether the solution space is discrete, continuous, or a mix of both.

The first-order nature of the theories means that we can use:

- Universal and existential quantifiers over variables
- Boolean connectives like $\wedge, \vee, \neg, \implies, \iff$
- Linear inequalities like $ax + by \leq c$
- Variables ranging over the appropriate number domain

It is crucial to highlight the linear restriction as we can not have:

- Multiplication between variables
- Non-linear operations

LRA and LIA, which is also known as Presburger Arithmetic, are decidable. That was shown in [Motzkin, 1936], and [Presburger, 1929]. The first-order formulation is what makes automated reasoning feasible. LA is simple and versatile and therefore used in different fields. For example, optimization for linear programming techniques, formal verification, and of course, automated reasoning. In automated reasoning, LA is mostly used in SMT solvers. For example, an SMT solver might prove that a program loop terminates for all possible inputs by reasoning about the linear relationships governing the loop's variables.

3.2 Challenges in Quantified Linear Arithmetic

LA could be seen as really simple, but it also presents some challenges. Finding solutions to LIA problems is computationally harder than solving problems in LRA, as integer solutions require more complex algorithms. If quantifiers $(\forall, \exists)$ are introduced, the complexity is further increased, as reasoning about quantified formulas requires advanced techniques like quantifier elimination. Additionally, the presence of uninterpreted function symbols adds another dimension of difficulty, because the reasoning algorithm must know all possible behaviors of these functions. Furthermore, scaling LA to handle large systems with many variables and constraints can also be computationally expensive.

3.3 Reasoning Methods in Linear Arithmetic

To reason in linear arithmetic, several methods could be used. Each method is designed to handle specific aspects of linear equations and inequalities in various contexts. For automated reasoning, methods aim to determine the satisfiability of formulas, verify properties, and solve constraints within automated systems. We can divide the methods into decision procedures including quantifier elimination, optimization-based methods, heuristic and approximation methods, and also some specialized methods. We will only focus on methods which are used for quantified LA.

In this thesis, the focus will be on three specific approaches in quantified LA:

- Superposition-Based Methods
- Instantiation-Based Methods
- Quantifier Elimination for Computer Algebra

These three categories cover the widely used reasoning methods for quantified formulas in LA. Superposition-based methods use mostly saturation algorithms. These can be found in [Cruanes, 2015], [Kovács and Voronkov, 2013] and [Schulz et al., 2019]. Essentially, the process begins with an initial set of clauses generated by preprocessing the input formulas (the starting search space). Inference rules, such as superposition, are then repeatedly applied to the clauses within this search space, and their resulting consequences (which are often non-ground) are added back into the search space. For Instance-based methods, repeatedly ground, quantifier-free, instances of quantified formulas are generated, and decision procedures are used to check satisfiability of the resulting set of ground formulas. These methods can be found in [Bonacina et al., 2017], [de Moura and Bjørner, 2007] and [Reynolds et al., 2017]. The two classes of methods are very different and can be seen as complements to each other. Further, we also need to mention an approach where quantifiers are eliminated „algebraically“. In the context of this thesis, such approaches will be called quantifier elimination for computer algebra, and are available in [Brown, 2003]

3.3.1 Quantifier Elimination

An algorithm for quantifier elimination removes all quantifiers of a formula φ until the quantifier-free formula ψ is equivalent to φ. Note that it could be enough that φ is equisatisfiable to ψ, that is φ is satisfiable if and only if ψ is satisfiable. We can say that a theory T admits quantifier elimination if there is an algorithm that given Σ-formula returns a quantifier-free Σ-formula φ that is T-equivalent. To get a better understanding, let's look at the following easy example.

Example 3.3.1 (Quantifier elimination)**.** Consider the formula $\exists a(a \approx x \wedge f(a) > 3)$. The quantifier-free formula $f(x) > 3$ is a result of a quantifier elimination of the formula.

Note that, for example, the formula $\exists x(f(x) > 3)$ has no quantifier elimination, since it is not possible to restrict f to have at least one value in its range that is greater than 3 without a quantifier.

Why do we need quantifier elimination? Simplification of logical formulas, by removing existential and universal quantifiers results in expressions that are often easier to analyze and also easier to solve. Complete quantifier elimination, even when possible, is computationally expensive. Therefore, most quantifier elimination methods approximate it, these approximations are often called quantifier reduction. If we approximate quantifier elimination, it is possible that some free variables may be left in the formulas after the transformation.

The first algorithm for quantifier elimination is called cylindrical algebraic decomposition (CAD) [Arnon et al., 1984], which is worst case doubly exponential in the number of variables. Quantifier elimination in applications outside of pure mathematics was fairly limited due to the practical complexity of the implemented methods for a long time. Some of these methods have been able to solve problems of interesting size in various fields, but most importantly in pure science. Due to the increase in computational power, mostly theoretical work contributed to that development. CAD has gone through numerous improvements, resulting in partial CAD and on the other hand, it has also been shown that real quantifier elimination is really hard for some problem classes, which turned the attention to certain problem classes and no standard procedures. Most approaches cope with formulas, where quantified variables are restricted to low degrees. For an introduction and more insight into CAD, we refer to [Jirstrand, 1995]. Further approaches for quantifier elimination and also some applications can be found in [Sturm, 2017].

There also exist many more techniques for quantifier elimination besides CAD. Variable elimination, for example, substitutes bound variables with terms or expressions. Also, skolemization is used for quantifier elimination, where existential quantifiers are replaced with function symbols. Resolution-based methods and term rewriting are also worth mentioning. There also exist virtual substitution methods. Furthermore, a quite prominent method is the Fourier-Motzkin elimination.

3.3.2 Fourier-Motzkin Elimination

Fourier-Motzkin Elimination (FME) [Fourier, 1827] is an algorithm used for solving systems of linear inequalities. The algorithm is used primarily in the context of LRA. Variables are eliminated to reduce a high-dimensional problem to a lower-dimensional one. At the core of it, FME uses variable projection. It removes one variable at a time from a system of inequalities, projecting the feasible region onto a lower-dimensional space, resulting in an equivalent system of inequalities in fewer variables.

The algorithm operates on a system of linear inequalities $\Sigma_{j=1}^{n} a_{ij}x_j \geq b_i,\ i = 1, \ldots, m$ where $a_{ij}, b_i \in \mathbb{R}$ for $i = 1, \ldots, m$ and $j = 1, \ldots, n$. It eliminates x_k for some

$k \in \{1, \ldots, n\}$ with the following steps:

1. For each $j \in \{1, \ldots, m\}$,
   - if $a_{jk} > 0$, multiply the j-th inequality by $\frac{1}{a_{jk}}$
   - if $a_{jk} < 0$, multiply the j-th inequality by $-\frac{1}{a_{jk}}$
2. Following, form a new system of inequalities:
   - take over all the inequalities in which the coefficient of x_k is 0
   - for every inequality, where x_k has a positive coefficient and for every inequality, where x_k has a negative coefficient, obtain a new inequality by adding them together

The first step is to guarantee that all nonzero coefficients of x_k are 1 or -1. In the second step, a new system is formed, which does not contain the variable x_k. Following, if we have $x_1^*, \ldots, x_n^*$ as a solution to the original system, $x_1^*, \ldots, x_{k-1}^*, x_{k+1}^*, \ldots, x_n^*$ is a solution to the new system. Furthermore, if $x_1^*, \ldots, x_{k-1}^*, x_{k+1}^*, \ldots, x_n^*$ is a solution to the new system, then there exists x_k^*, such that $x_1^*, \ldots, x_n^*$ is a solution of the original system. Well, this states that the original system has a solution if and only if the new system has a solution. The FME should be applied repeatedly to get a system with at most one variable such that it has a solution if and only if the original system has one. Solving systems of linear inequalities with at most one variable is simple, so we can come up with whether or not the original system has a solution.

SMT solvers use FME for reasoning about LRA. The big advantage of FME is the simplicity of implementation, and it provides precise results for systems of linear inequalities. But, FME also has some limitations, as the number of inequalities can grow exponentially with each elimination step, and of course, FME only applies to linear inequalities in LRA.

3.3.3 Superposition Calculus

The Superposition Calculus is a refutationally complete inference system, which is used for automated reasoning in FOL with equality. It was introduced in [Bachmair and Ganzinger, 1994] and is more or less the foundation of saturation-based theorem proving. The calculus extends ordered paramodulation by integrating rewriting and simplification techniques to handle equational reasoning efficiently. Ordered inference rules with selection functions are used for controlling derivation. The Superposition Calculus is refutationally complete, which means that it can derive a contradiction if the clause set is unsatisfiable. Saturation can be seen as a key property, as it ensures that all non-redundant inferences are generated.

The calculus is, in some forms, implemented in modern theorem provers like E, Vampire and SPASS. There exist several extensions of the Superposition Calculus to enhance its reasoning capabilities.

3.3.4 Coopers Algorithm

Coopers algorithm is a quantifier elimination procedure for Presburger arithmetic (PA). PA is a decidable theory, focusing on FOL of natural numbers $\mathbb{N}$ with addition and equality as operations. Mojżesz Presburger introduced it in 1929 [Presburger, 1929]. We can define PA as the set of those sentences that are true in the interpretation with the structure of non-negative integers, including the constants 0, 1, and the addition of non-negative integers. Note that multiplication of variables is not included in PA.

Cooper introduced a decision procedure for PA in [Cooper, 1972]. It eliminates existential quantifiers one by one, while logical equivalence is preserved. Consider a formula $\exists x F(x)$, where F is quantifier-free. The idea of the algorithm can be generally described in the following steps:

1. Put formula on negation normal form (NNF), getting $F_1(x)$
2. Normalize $F_1(x)$ to use $<$ as the only comparison operator, getting $F_2(x)$
3. Normalize $F_2(x)$ so that atomic formulas have one occurrence of x, getting $F_3(x)$
4. Normalize $F_3(x)$, such that the coefficients of x is 1, getting $F_4(x')$
5. From the formula $F_4(x')$ a quantifier free formula F_5 can be produced, which is equivalent to $\exists x F(x)$

The algorithm itself is not very efficient, as it runs in doubly exponential time in the worst case. For large formulas, the algorithm is really inefficient, such that in practice often alternatives are used, however it is theoretically important as a complete decision procedure for PA.

3.3.5 Counterexample-Guided Instantiation

Counterexample-Guided Instantiation (CEGI) is a quantifier instantiation technique used in SMT solving and automated reasoning, particularly for quantified formulas over theories like LIA and LRA.

Instead of just randomly generating all possible instances of a quantified formula, with CEGI, one can choose instantiations based on counterexamples from a model that violates the current partial approximation of the formula. That process happens in a feedback loop-like manner. First of all, a candidate model is guessed and then a check happens whether the current model violates any quantified formula or not. If it violates the formula, the quantified formula is instantiated with the values from the model. In the end, the loop is repeated and the set of instances is refined until the formula is proven valid or a counterexample is found.

CEGI may fail to terminate if it is not guided carefully or if the theory itself is undecidable, but if it terminates, it avoids combinatorial explosion of full quantifier instantiation. An advantage of this approach is that it works very well with LA and uninterpreted functions. The performance depends on heuristics, where one needs to use effective term selection functions and so on. Further, it can also be used to extract witnesses for models.

3.3.6 Syntax-Guided Instantiation

Syntax-Guided Instantiation (SyGI) is a quantifier instantiation technique used in SMT solving, where quantified formulas are instantiated by generating terms whose syntax matches expressions seen in the formula or the current model. This is in contrast to just trying all possible values (mostly impossible). SyGI uses the formulas' syntax itself to guide which values to try and which might be useful.

3.3.7 Virtual Substitution

Virtual Substitution [Weispfenning, 1997] is a technique for eliminating quantifiers from logical formulas. Other than computing a full quantifier-free equivalent formula like Fourier–Motzkin or CAD, Virtual Substitution works by substituting expressions for the quantified variables. The technique simulates the effect of quantification without full enumeration.

The candidates for the substitution are most of the time called test terms. These expressions need to be identified by Virtual Substitution in the first step and could be roots of polynomials, infinitesimals or numeric constants. Virtual Substitution substitutes each test term into the formula, and each substitution gives a case that has to be checked.

This approach is really efficient for LA and avoids heavy computation efforts, which are needed for example for CAD. Note that Virtual Substitution can only be used for LRA and not for LIA.

3.3.8 Model-Based Projection

Model-Based Projection (MBP) [Komuravelli et al., 2014] eliminates existentially quantified variables from a formula using models. The models are used to guide the simplification of the formula. Instead of computing a full quantifier elimination, MBP uses models to identify a satisfying assignment of a formula. Further, MBP extracts a simpler formula, which over-approximates quantifier elimination. MBP can be used both in LRA and LIA.

CHAPTER 4

Quantified Reasoning in Linear Arithmetic

Reasoning in quantified linear arithmetic is a key challenge in automated reasoning. In the following sections, background information and important papers in three major approaches for reasoning in quantified linear arithmetic are summarized and compared.

If a logical statement is evaluated with quantifiers over variables constrained by linear relationships, such as linear equations and inequations, we refer to that as quantified reasoning in LA. With quantifiers, it is possible to express more complex properties and constraints. We use quantified reasoning in LA for various applications like formal verification, model checking and constraint solving. To solve most of these problems in practice, we need to transform formulas into equivalent quantifier-free forms with some specific techniques and approaches.

4.1 Quantifier-Free Reasoning in Linear Arithmetic

LA involves equations and inequalities where variables appear with at most degree one. When quantifiers, such as $\forall$ and $\exists$ are absent, reasoning methods can focus on directly determining the satisfiability constraints without using computationally expensive quantifier elimination. In the following, some background information and alternative approaches are presented. Note that the quantifier-free fragment is not the main focus of this thesis, and therefore, these approaches are not introduced in detail.

4.1.1 Integrating Linear Arithmetic into Superposition Calculus

The paper [Korovin and Voronkov, 2007] introduces the Linear Arithmetic Superposition Calculus ($LASCA$). The authors extend the superposition calculus with rules for rational LRA with equality and Fourier Motzkin Elimination for reasoning with inequalities. The

extension works with rules similar to superposition and ordered chaining rules in first order reasoning. Existing approaches have some limitations that rely on approximations or incomplete methods when dealing with LRA, which the authors wanted to address. These approaches are mostly based on an approximation of arithmetic reasoning by considering an axiomatisable theory [Godoy and Nieuwenhuis, 2004, Stuber, 1998, Waldmann, 2001, Waldmann, 2002]. The biggest challenge in integrating LRA with superposition calculus is the computational complexity of handling rational numbers ($\mathbb{Q}$) and the interaction between theory terms and other non-theory terms. Unlike previous approaches, which rely on approximations or incomplete methods, *LASCA* operates directly with the theory $\mathbb{Q}$ and therefore avoids abstraction techniques that introduce additional variables, leading often to inefficiencies.

The *LASCA* calculus extends the traditional superposition calculus by introducing rules tailored for LA. One rule considers Gaussian Elimination, with handling equations by substitution of one variable in terms of others. Another rule addresses Fourier Motzkin Elimination, where inequalities are chained to eliminate variables and to simplify constraints. Also, Inequality Factoring is a rule, where combined inequalities are simplified in order to identify contradictions or refinements. There is also a rule for Ordering and Normalization, which ensures that clauses are in a canonical form for efficient comparison. The *LASCA* calculus for ground clauses is sound, which is shown by derivation rules that maintain consistency with the theory $\mathbb{Q}$. *LASCA* completeness is conditional, because a saturated set of clauses needs to satisfy additional constraints to ensure that satisfiability is decidable.

For handling non-ground clauses, i.e. clauses with variables, the *LASCA* calculus introduces lifting techniques. To prevent undecidable interactions between theory and other non-theory terms, variables are restricted to „safe sorts“. The calculus incorporates rules for managing variable occurrences, using normalization and associativity and commutativity unifiers (AC-unifiers). In order to work with non-ground terms, Gaussian and Fourier Motzkin Elimination are generalized, to ensure soundness of the lifted rules.

To ensure termination, $\mathbb{Q}$-Knuth-Bendix ordering ($\mathbb{Q}$-KBO) is introduced. This ordering is based on the Knuth-Bendix ordering and it is also AC-compatible, monotonic and also satisfies finiteness conditions, which are crucial for proving completeness. Furthermore, the authors prove that the general validity problem for FOL with LA is Π^1_1-complete. This result implies that no complete calculus exists without finiteness constraints, and therefore, the existence of practical limitations of reasoning systems for LRA with non-theory functions is highlighted. The paper itself serves as a foundation for combining FOL with arithmetic reasoning, which is then further picked up by [Korovin et al., 2023] to introduce the *ALASCA* calculus, and also shows the potential for practical theorem proving tasks.

Other approaches

Quantifier-free reasoning in LA is a well-studied area, and several standard approaches and algorithms are known and implemented in automated reasoning tools. For LRA and LIA there exist several specialized decision procedures that efficiently determine the satisfiability of quantifier-free formulas. Examples of procedures are the Simplex Method, Cutting Plane Methods, Branch and Bound Methods, Fourier–Motzkin Elimination, Presburger Arithmetic Decision Procedures, CDCL(T) + Theory Propagation and Difference Logic Solvers, which can be found in following works: [Dutertre and de Moura, 2006], [Stansifer, 1984], [Ferrante and Rackoff, 1975], [Reddy and Loveland, 1978], [Nelson and Oppen, 1979].

4.2 Superposition-Based Methods

4.2.1 ALASCA: Reasoning in Quantified Linear Arithmetic

Due to the efficient superposition calculus, first-order theorem provers perform very well on quantified problems, but support for arithmetic reasoning is limited to heuristic axioms. The authors introduce in [Korovin et al., 2023] the *ALASCA* calculus, which lifts superposition reasoning to the LA domain. They also show that *ALASCA* is both sound and complete with respect to an axiomatisation of LA, while proving first-order quantified LA properties.

The superposition calculus is a refutationally complete calculus for first order logic with equality. A natural solution to the inefficieny of theory reasoning via superpositionr is, to try to eliminate some theory axioms, but this is very difficult in theory and practice. The *LASCA* calculus [Korovin and Voronkov, 2007] replaced several theory axioms of LA, including transitivity of inequality, by a new inference rule inspired by Fourier-Motzkin elimination and also some additional rules. *LASCA* is complete for the ground case, however *LASCA* was not implemented. This is due to its lack of clear treatment for the non-ground case, and of course, also complexity. The authors of [Korovin et al., 2023] introduce a new non-ground version of *LASCA*, which is called *ALASCA*. The *ALASCA* calculus comes with new abstraction mechanisms, inference rules and orderings, which together are proven to yield a sound and complete approach with respect to a partial axiomatisation of LA. This is done by introducing a novel variable elimination rule within saturation-based proof search, an analogue of unification with abstraction needed for non-ground reasoning, and a new non-ground ordering and powerful background theory for unification, which is not restricted to arithmetic but can be used with arbitrary theories, resulting in, that *ALASCA* does work and scales. Therefore, *ALASCA* improves *LASCA* by ground modifications and lifts *LASCA* in a finitary way. Furthermore, *ALASCA* is implemented in Vampire, and it is shown that it solves overall more problems than existing theorem provers.

ALASCA extends *LASCA* with unification with abstraction (uwa). Uwa is needed because first-order arithmetic reasoning requires establishing syntactic difference among

terms (e.g. $4x$ and $x - 1$), while deriving, there are some instances that are semantically equal in models of a background theory $\mathcal{E}$. This problem can be addressed in a naive way by using an axiomatisation of the background theory $\mathcal{E}$ and then using this axiomatisation for proof search in uninterpreted FOL. That approach can be very costly and therefore, to circumvent such inefficient handling of equality reasoning, one can use unification modulo $\mathcal{E}$ [Kapur and Narendran, 1992, Reger et al., 2018, Waldmann, 1998]. The authors of [Korovin et al., 2023] made some adjustments towards unification modulo $\mathcal{E}$, introducing unification with abstraction. Furthermore, they also show how a complete superposition calculus using unification modulo $\mathcal{E}$ can be turned into a complete superposition calculus using unification with abstraction.

To put it simply, unification modulo $\mathcal{E}$ finds substitutions σ that make two terms s, t equal in the considered background theory. More formally, this means $\mathcal{E} \models s\sigma \approx t\sigma$. But there are some inefficiencies, most importantly, there is no unique most general unifier $mgu(s, t)$ when unifying modulo $\mathcal{E}$, but only minimal complete sets of unifiers $mcu_{\mathcal{E}}(s, t)$, which can be very large. The presented uwa method can be seen as a lazy approach of full abstraction from [Waldmann, 1998]. The abstracting unifiers uwa(s, t) $= <\sigma, \mathcal{C}>$ are computed to replace unification modulo $\mathcal{E}$ by unification with abstraction. These results are then used to introduce the $ALASCA$ calculus for reasoning in quantified arithmetic, by combining superposition reasoning with Fourier-Motzkin type inference rules [Korovin et al., 2023].

There is also an altered ground version $ALASCA^{\theta}$, that can be efficiently lifted to the quantified domain. $ALASCA^{\theta}$ operates on clauses and employs a partial axiomatisation $A_{\mathbb{Q}}$ of rational number models ($\mathbb{Q}$-models). This divides axioms into two categories: equality axioms (A_{eq}) and inequality axioms (A_{ineq}). Following, the key inference rules in $ALASCA^{\theta}$ are listed:

- Fourier-Motzkin Elimination (FM): Chains inequalities by eliminating variables.
- Tight Fourier-Motzkin Elimination (FM$^{\geq}$): Similar to FM, but handles cases where inequalities include equalities.
- Inequality Factoring (IF): Simplifies combinations of inequalities with the same term.
- Superposition (Sup): Extends standard superposition techniques to arithmetic terms, replacing terms based on axioms or previously derived equalities.

$ALASCA^{\theta}$ relies on a normalization technique, which is called A_{eq}-normalization. Terms are simplified into a canonical form to reduce the search space during inference. During proof construction, an A_{eq}-compatible ordering ensures a consistent prioritization of terms. The main advantages of $ALASCA^{\theta}$ are that it simplifies proof search by eliminating redundant or non-ground instances and it efficiently integrates arithmetic reasoning with

general-purpose theorem proving.

Lifting $ALASCA^{\theta}$ to the quantified domain involves extending its reasoning to handle clauses containing quantifiers and non-ground terms. The main challenge here is: dealing with variables that appear as top-level terms, called unshielded variables. To address this, $ALASCA$ introduces a Variable Elimination rule (VE), which converts any clauses containing such variables into an equivalent set of clauses without these variables. Lifting to the quantified domain also integrates unification with abstraction (uwa). With ground completeness and also the lifting to the quantified domain, $ALASCA$ is complete, namely refutationally complete with respect to $A_{\mathbb{Q}}$ for sets of clauses without unshielded variables. ALASCA was implemented in the theorem prover Vampire [Kovács and Voronkov, 2013].

4.2.2 Superposition Modulo Linear Arithmetic SUP(LA)

The paper [Althaus et al., 2009] introduces SUP(LA), a hierarchic superposition based theorem proving calculus for FOL with LA. This approach instantiates the hierarchic theorem proving approach SUP(T) for any theory T[Bachmair et al., 1994] to SUP(LA). This offers an abstract completeness result for the combination via a sufficient completeness criterion that goes beyond ground problems, which is needed for DPLL(T). It also enables the handling of LA in a modular way and could be implemented via efficient off the shelf solvers. The approach can even decide satisfiability of first-order theories with universal quantification modulo LA.

The SUP(LA) calculus operates within a many-sorted logical framework with a base sort, which is reserved for LA terms over the rationals and other kinds, which represent non-arithmetic terms. Function symbols are divided into function symbols interpreted in the base theory LA and free function symbols for non-arithmetic operations. The SUP(LA) calculus has a hierarchical specification where FOL over free function symbols is combined with a base theory over LA. Note that this allows reasoning to be modular, where base constraints are solved using LA decision procedures and first-order literals are handled by superposition rules. The authors defined a clause form that separates base constraints and first-order literals. A clause is represented as $\Lambda||\Gamma \rightarrow \Delta$, where Λ contains base constraints from LA and Γ, Δ contains first-order literals using free function symbols. To extend the calculus for the general case of a hierarchic specification [Bachmair et al., 1994], any given disjunction of literals can be transformed into a clause of the form $\Lambda||\Gamma \rightarrow \Delta$. If a subterm t, whose top symbol is a base theory symbol, occurs immediately below a non-base symbol, it is replaced by a new base sort variable u, and this is added to Λ. Also, if a subterm t, whose top symbol is not a base theory, occurs immediately below a base operator symbol, it is replaced by a general variable x and is added to Γ. These transformation steps are repeated until all terms in the clause are pure, which results in all base literals being in Λ and all non-base literals being in Γ, Δ. Note that Γ, Δ is defined as sequences of atoms where Λ holds theory literals. Clauses need to be „purified“ only once before saturating the clauses, and if the premises of an inference are abstracted clauses, then the conclusion is also abstracted. SUP(LA) uses

hierarchic superposition rules extended to handle LA constraints. The main inference rules are the following:

- Superposition Left
  - A left-hand side term in an equation is replaced with its right-hand side in another clause.
  - This ensures most general unification and maximality criteria for efficiency.
- Superposition Right
  - This is more or less similar to Superposition Left, but applies to the right-hand side of an equation.
- Equality Factoring
  - This rule simplifies equalities by combining multiple equations.
- Ordered Factoring
  - Combines two equal literals in a clause by unification.
- Equality Resolution
  - If both sides unify, this rule eliminates equations.
- Constraint Refutation
  - That rule checks for unsatisfiability of clause constraints in the base theory LA.

With these inference rules, first-order reasoning is combined with LA constraint solving and allowing for modular reasoning in the hierarchical setting. Furthermore, effective redundancy criteria specific to SUP(LA) are introduced, where hierarchical redundancy is mapped to linear programming problems for satisfiability subsumption and implication tests. A sufficient completeness criterion ensures that ground instances are sufficiently complete for the base sort. Most importantly, if the base theory is compact, then SUP(LA) is refutationally complete. This ensures completeness for SUP(LA).

The paper [Althaus et al., 2009] also discusses efficient algorithms for constraint solving in LA with three specific tasks: Satisfiability Checking, Implication Test and Subsumption Test. These tasks are mapped to linear programming problems. For that, Farkas' Lemma [Farkas, 1902] and its extensions to strict and non-strict inequalities are needed. Farkas' Lemma is a fundamental result in linear algebra and optimization theory that provides a condition for the feasibility of linear inequalities. Farkas' Lemma initially applies to non-strict inequalities. To handle strict and non-strict inequalities, the lemma is extended using transformations. For more information, see for example

[Matoušek and Gärtner, 2007]. The implication test checks, given two sets of constraints Λ_1 and Λ_2, if: $\Lambda_2 \subseteq \Lambda_1$. The subsumption test checks, given two clauses with constraints Λ_1 and Λ_2, if Λ_1 subsumes Λ_2. All tests can be reduced to linear programming feasibility problems and can therefore be solved using existing linear programming solvers. The SUP(LA) calculus is implemented as an extension of the SPASS theorem prover, SPASS(LA). SUP(LA) preserves quantifiers through the reasoning process and enables quantified reasoning.

4.2.3 Beagle – A Hierarchic Superposition Theorem Prover

The paper [Baumgartner et al., 2015] describes the automated theorem prover Beagle. Beagle itself implements the hierarchic superposition calculus. This calculus is used for automated reasoning in a hierarchic combination of FOL and a background theory. The background theories implemented are LIA and LRA. The prover features new simplification rules for theory reasoning and also implements calculus improvements such as weak abstraction, and determining satisfiability and unsatisfiability with respect to quantification over finite integer domains.

Beagle extends the hierarchic superposition calculus [Bachmair et al., 1994]. The hierarchic superposition calculus implements hierarchic superposition in a hierarchic combination of FOL and background theories (BG). A BG is defined by a BG signature $\Sigma_B = (\Xi_B, \Omega_B)$ where Ξ_B is defined as BG sorts, which would be for example *int* for LIA and Ω_b defines BG operators. Then there are BG clauses that use BG sorted variables and are passed to the BG prover to check for satisfiability. Further, there is also a Foreground theory FG, which is defined as $\Sigma = (\Xi, \Omega)$, where $\Xi_F = \Xi \setminus \Xi_B$ and $\Omega_F = \Omega \setminus \Omega_B$. FG terms themselves contain at least one FG operator or FG sorted variables. The intended semantics are conservative extensions of BG specifications, preserving the interpretation of BG sorts and operators. Weak abstraction abstracts out BG terms other than variables and number constants that occur as subterms of FG terms. With the inference rules of hierarchic superposition, the FG prover saturates the set of Σ-clauses. This happens, for example, with negative superposition, which only applies to the FG parts of clauses and uses weak abstraction for the conclusion. A requirement for the refutational completeness of hierarchic superposition is sufficient completeness. That means that every ground BG sorted term is equal to some BG term. To ensure sufficient completeness, the define rule is used, which introduces a new parameter for BG sorted FG terms, and therefore maintains refutational completeness. BG reasoning is represented in Beagle as theory specific modules, which are called solvers, and they implement a specific interface. B-Satisfiability is the satisfiability of a clause constraint in the BG, which means that the BG literals in the constraint are satisfiable under the interpretation of the background theory.

In Beagle, every solver needs to provide a decision procedure for B-satisfiability of sets of BG clauses. The solver interface supports quantifier elimination for eliminating variables occurring only in BG literals. It also supports the splitting of BG clauses into variable disjoint subclauses and simplification with cautious and aggressive rules,

resulting in sufficient and refutational completeness. Beagle implements solvers for LIA and LRA. For LIA, quantifier elimination is implemented using Cooper's algorithm with some performance improvements. Further, simplification and arithmetic terms normalization is used. Cautious simplification preserves refutational completeness and aggressive simplification optimizes performance, but as a trade off risks completeness. For LRA, the solver implements a Fourier-Motzkin style quantifier elimination procedure. The decision procedure uses the Simplex algorithm [Dutertre and de Moura, 2006], which is further extended to strict inequalities. Also, here, cautious and aggressive simplification is supported.

The proof procedure of Beagle uses standard techniques, but treats BG formulas in a specific way. The prover uses two translators to convert formulas into CNF. For existentially quantified variables, skolemization is used, and integer variables are handled by applying quantifier elimination when possible. For universal BG formulas over integers, Skolem functions are not introduced. The main loop of Beagle uses the Discount saturation loop. The loop maintains two clause sets: Old and New. New contains clauses that have not yet been expanded, and Old contains clauses that have already been processed. Simplified, the prover picks a clause from New, moves it to Old, and applies inference rules to expand it while keeping background constraints and preserving quantifiers. Further, new clauses are simplified.

4.3 Instantiation-Based Methods

4.3.1 Solving Quantified Linear Arithmetic by Counterexample-Guided Instantiation

The framework presented in the paper [Reynolds et al., 2017] introduces instantiation-based decision procedures for verifying the satisfiability of quantified formulas. These procedures work for LIA and LRA with one quantifier alternation. The authors use a unique strategy to address quantified formulas, and these techniques can be integrated into standard SMT solvers.

The conventional method for determining constraints over quantified theories is quantifier elimination, which aims to convert arbitrary quantified formulas with free variables into an equivalent form without quantifiers. Nevertheless, the complexity of performing quantifier elimination is really high. SMT solvers frequently adopt instantiation-based approaches because, even if they are incomplete, they perform strongly on handling undecidable fragments of FOL. The authors introduce a novel method for verifying the satisfiability of formulas in quantified LA through a fresh quantifier instantiation framework. Quantifier instantiation in this context is used because:

- Techniques that rely on lazy quantifier instantiation can check satisfiability significantly faster than their theoretical computational complexities suggest

- Using quantifier instantiation for decidable fragments enables seamless integration and synergy with existing instantiation-based methodologies, which are used by state-of-the-art SMT solvers
- A big subset of synthesis problems can be formulated as quantified formulas featuring one quantifier alternation

Recent efforts showed that leveraging quantifier elimination in a more efficient and practical manner, focusing on generating equisatisfiable ground formulas, is sometimes more efficient. The use of quantifier instantiation with decidable fragments in quantified LA not only speeds up satisfiability checks but also enables a good incorporation of existing instantiation-based frameworks.

The introduced framework operates based on instantiation, meaning it involves substituting specific values (instantiations) for quantified variables within the formulas to evaluate the satisfiability of those. The goal of the framework is to develop decision procedures that can effectively determine whether a given quantified formula is satisfiable within the given constraints of the specified theory. The idea is to incrementally instantiate quantified variables with concrete values, guided by counterexamples or selection functions. The general high level workflow for instantiation-based procedures can be seen as follows:

1. Start with a quantified formula.
2. Initialize an empty set of instances.
3. Use a selection function to select candidate values for quantified variables.
4. Use these values to instantiate the formula to convert it to a quantifier-free form.
5. Check the quantifier-free formula using a standard SMT solver.
6. Refine the instances using counterexamples until no new instances are needed.

The approach avoids full quantifier elimination by lazily instantiating only relevant quantifiers. The general procedure is proven sound by maintaining logical equivalence between the original quantified formula and its instantiated versions. Termination is guaranteed when the selection function is finite, monotonic and model-preserving. For LRA and LIA with one quantifier alternation, the paper presents selection functions S_{LRA} and S_{LIA} that are finite and monotonic, ensuring the completeness and termination of the instantiation-based procedure. These selection functions play a crucial role in the instantiation-based procedure and are inspired by the Loos-Weispfenning [Loos and Weispfenning, 1993] or Ferrante-Rackoff [Ferrante and Rackoff, 2006] methods for quantifier elimination, but avoid the need for computing virtual terms. The key aspects are the finiteness and monotonicity of these selection functions, which are fundamental for ensuring both completeness and termination of the procedure. The

finiteness of the functions implies that there exists only a finite number of possible tuples that the functions can return for any given interpretation, set of formulas, variables, and additional parameters. Further, this property is essential for the computational efficiency of the instantiation-based procedure and guarantees that the procedure will not get stuck in an infinite loop. In contrast, monotonicity is important for ensuring that the selections made during the runtime do not violate a certain order or constraint relationships. A selection function is considered monotonic if it preserves certain relationships between the input formulas and the selected terms during each iteration. Monotonicity is needed for soundness of the instantiation-based decision procedure. To instantiate LRA, the selection function S_{LRA} uses a distinguished constant δ to instantiate real variables. That ensures that all possible values within an interval are covered by choosing boundary points and infinitesimal shifts. Unlike Fourier-Motzkin Elimination or Ferrante-Rackoff, which need explicit case splitting, this method is more efficient because it lazily instantiates only relevant cases. On the other hand, to instantiate LIA, the selection function S_{LIA} uses integer division rounding to handle integer constraints. The instances themselves are chosen by rounding up and down to the nearest integers and using congruence modulo constraints to ensure all classes are covered. Lazy instantiation is combined with modulo reasoning and so, the number of necessary instances is reduced. One could think of extending Cooper's algorithm in that case, but other than explicit case splitting, all disjuncts are enumerated. The selection functions are sound and complete for both LRA and LIA with one quantifier alternation. Furthermore, also instantiation for LIRA is discussed. Therefore, conversion functions are introduced for rounding real numbers to integers. The selection function itself is kind of a hybrid. It firstly instantiates real and, after that, integer variables and ensures compatibility and consistency between real and integer constraints. In practice, this method is helpful, but note that completeness is not guaranteed in this case. Furthermore, the framework is extended to handle arbitrary quantifier alternations. In that case, the method does not require the formulas to be in prenex form. Also, nested instantiation loops are used to incrementally instantiate quantifiers while preserving the logical structure. The introduced instantiation-based approach is also integrated within the SMT solver CVC4 [Barrett et al., 2011].

4.3.2 Theory Instantiation

The paper [Ganzinger and Korovin, 2006] explores the concept of theory instantiation, which can be described as the process of applying a general theory to a specific case or a specific instance. The authors develop a method for integrating theory reasoning into an instantiation-based framework.

For integrating theory reasoning into a logical calculus, two approaches are mostly used: black-box and glass-box approaches. For the glass-box approach, theory reasoning is integrated with specific inference rules. The resulting calculus is very efficient for a specific theory, but one needs to come up with rules for each theory, which can make completeness arguments for the calculus non-trivial. For integrating various theories into the resolution based framework, there is a lot of literature, but

for the case of integration into the instantiation framework beyond the integration of equality reasoning, not really much is known. More information can be found in [Ganzinger and Korovin, 2004, Baumgartner and Tinelli, 2003, Letz and Stenz, 2002].

The paper [Ganzinger and Korovin, 2006] introduces theory instantiation, which is closely related to theory resolution and can be seen as a black-box approach. The approach differs from others in a way that it allows to employ off-the-shelf satisfiability solvers for ground clauses modulo theories. Such reasoners are used for many important theories, and some very efficient implementations are available [Barrett et al., 2005].

[Ganzinger and Korovin, 2006] introduce an instantiation-based inference process for reasoning modulo a universal theory T. The process starts with a given set of first-order clauses S and then all variables in all clauses in S are mapped into the distinguished constant $\bot$, obtaining a set of ground clauses $S\bot$. If $S\bot$ is T-unsatisfiable, then S is also T-unsatisfiable, and the process stops. If that is not the case, a literal in each clause gets selected non-deterministically. From that set of literals $\mathcal{L}$ is obtained. If $\mathcal{L}$ is T-satisfiable, then S is also T-satisfiable and the process stops. Otherwise, relevant instances of clauses from S are generated, witnessing T-unsatisfiability of $\mathcal{L}$ at the ground level. This is all done based on the Unit Calculus (UC). To obtain refutational completeness of the process, it needs to be ensured that sufficiently many instances of clauses are generated. For that, UC is required to be answer-complete. In the end, all obtained instances of clauses are added to S.

The theory reasoner itself needs some requirements, which will be done in terms of UC. The calculus is used for proving T-unsatisfiability of sets of literals and that also provides substitutions to generate relevant instances witnessing T-unsatisfiability. Further, the authors introduce the notion of answer-completeness, which is needed for overall completeness of the instantiation process. To put it simply, answer completeness ensures UC can generate substitutions to detect inconsistencies in sets of literals modulo a theory T.

The main contribution of [Ganzinger and Korovin, 2006] is the instantiation calculus TInst-Gen. The authors show that if a set of clauses S is saturated with respect to TInst-Gen, either $S\bot$ is already T-unsatisfiable and a theory reasoner for ground clauses can detect the unsatisfiability, or otherwise S is T-satisfiable. The inference system will be guided by a selection function sel. This function is used on clauses which will be based on a model for the ground clauses $S\bot$. Further, the semantic notion of redundancy from [Ganzinger and Korovin, 2003] is adapted. The authors state that a set of clauses S is called TInst-saturated up to redundancy with respect to a selection function sel if all inferences in TInst-Gen with premises from S are T-redundant in S. It is also shown how TInst-Gen saturation of a set of clauses can be achieved as a limit of a fair saturation process.

Furthermore, the approach of [Ganzinger and Korovin, 2006] to theory reasoning is also suitable for combining the instantiation calculus with other calculi. This is really interesting. The goal is to divide the set of input clauses into two classes. One class can be taken as theory clauses and a specialized procedure is applied to them. The other classes is the set of clauses, which are treated with the instantiation calculus. The theory reasoner can be a logical calculus itself.

A big contribution of [Ganzinger and Korovin, 2006] is a theorem, which implies that if the theory reasoner satisfies the requirements, then any fair instantiation process is complete for reasoning modulo the theory. The shown process can be guided by information on models for ground clauses. Also, the presented framework allows to justify redundancy elimination based on a notion of redundant clause and inference.

4.3.3 Syntax-Guided Quantifier Instantiation

The paper [Niemetz et al., 2021] presents an approach for quantifier instantiation in SMT. In detail, syntax-guided quantifier instantiation is used for various theories, most interestingly for LIA and LRA. The authors mention as motivation that current approaches lack generality, as they require theory-specific selection functions. Furthermore, this challenge increases, considering reasoning over combined theories. Syntax-guided synthesis is also introduced, which is used to select instantiation terms through a grammar-based approach. As stated before, this works with LIA or LRA but also for floating point arithmetic and fixed-size bit-vectors.

The presented approach combines counterexample-guided quantifier instantiation with enumerative syntax-guided synthesis. This relies on grammar-based instantiation. Each quantified variable needs to be associated with a syntax-guided synthesis grammar based on its type (LIA, LRA, etc). The grammar itself defines the set of terms for instantiation. The paper introduces an algorithm for syntax-guided quantifier instantiation that tries to synthesize a term t for a variable x in a given formula $\forall x P(x)$ such that $\neg P(t)$ holds. The idea of the algorithm can be seen as follows:

1. For each quantified variable, a datatype grammar is created.
2. Counterexamples need to be considered, which falsify the quantified formula.
3. Using smart enumerative syntax-guided synthesis terms are enumerated from the grammar.
4. Lemmas are generated incrementally, where quantified formulas are instantiated with the generated terms and therefore, consistency between the datatype grammar and the term interpretations is ensured.

The authors of [Niemetz et al., 2021] show that the algorithm is sound and therefore guarantees, if unsat is returned, the quantified formulas are unsatisfiable, and also if

sat is returned, the formulas are satisfiable. The algorithm continuously refines the model, ensuring progress toward a solution. For the grammar construction itself, syntax-guided quantifier elimination provides default grammars for all supported types. As traditional approaches use mostly „pattern matching“ or „model-based instantiation“, here a „grammar-based approach“ is used. Each quantified variable is associated with a datatype grammar, which defines the set of terms for instantiation, and it is also possible to use custom grammars. Terms are selected for grammar construction based on two strategies: scope-based and size-based. Scope-based strategies determine the origin of terms used for instantiation. Syntax-guided quantifier elimination defines three scopes: *in*, *out* and *both*. To select terms from the quantified formula, *in* is used. To select terms from the set of ground formulas, *out* is used, and to combine terms from both sets, *both* is used. Size-based strategies prioritize terms based on their syntactic size, and therefore allow for a step-wise exploration of the term space. Here, *min*, *max* and *both* are considered. To select minimal subterms, focusing on constants or variables, *min* is used. To select maximal subterms, exploring complex terms first, *max* is used and further, to combine minimal and maximal subterms for a balanced exploration, *both* is used. It is possible to come up with any combination of scope-based and size-based strategies. For example *in-both*, *in-min*, *out-max* or *both-both*. Syntax-guided quantifier elimination also introduces two types of lemmas: instantiation lemmas and evaluation unfolding lemmas. Instantiation lemmas are derived from the quantified formula by substituting terms generated from the grammar. Evaluation unfolding lemmas ensure that the instantiated terms are correctly interpreted according to the datatype grammar. Also, lemmas need to be selected using a procedure. There exist three selection techniques: *interleave*, *priority-inst*, *priority-eval*. To alternate between instantiation lemmas and evaluation unfolding lemmas, *interleave* is used. To prioritize instantiation lemmas, as the name suggests, *priority-inst* and to prioritize evaluation unfolding lemmas, *priority-eval* is used.

The new syntax-guided synthesis based instantiation approach combines counterexample-guided quantifier instantiation with smart enumerative syntax-guided synthesis techniques to synthesize terms for quantifier instantiation. Further, the syntax-guided quantifier instantiation was implemented by the authors in CVC4 [Barrett et al., 2011].

4.4 Quantifier Elimination in Computer Algebra

4.4.1 VIRAS: Conflict-Driven Quantifier Elimination for Integer-Real Arithmetic

The paper VIRAS: Conflict-Driven Quantifier Elimination for Integer-Real Arithmetic [Schoisswohl et al., 2024] introduces the so called VIRAS method, a quantifier elimination procedure for LIRA. The authors address challenges in solving quantified problems in LIRA, as traditional approaches struggle with efficiency and scalability for LIRA problems. VIRAS itself combines virtual substitutions, conflict-driven proof search and Cooper's method to deliver exponential speed-ups over existing approaches.

The VIRAS approach generalizes Cooper's method for LIA to LIRA. Virtual substitutions in VIRAS make the systematic replacement of variables in the problem with complex expressions involving other variables possible, allowing to transform the original problem into a more tractable form. With the conflict-driven proof search mechanism, it is possible to explore all possible substitutions and constraints, needed to identify contradictions or conflicts that can lead to efficient problem solving. The generalization of Cooper's method for LIRA enhances the precision and effectiveness of handling integer constraints within the problem. The primary contribution of VIRAS is its ability to offer an exponential speedup to existing methods for quantified arithmetic reasoning. By combining the three key components, VIRAS demonstrates performance in solving complex quantified arithmetic problems that are beyond the capabilities of traditional SMT-based techniques.

The VIRAS method translates a quantified formula $\exists x\phi$ into an equivalent quantifier-free formula ϕ', where if all variables in ϕ are bound, ϕ' is ground and can easily be evaluated. Universal quantifiers are expressed in terms of existential ones, and existential quantifiers can be distributed over disjunctions. Note that this work considers fixed $\exists x\phi$ formulas, where ϕ is a conjunction of literals containing free variables considered parameters. The basic idea of VIRAS is to compute a finite but sufficient number of witnesses for $\exists x$ and convert the quantified formula $\exists x\phi$ into an equivalent finite disjunction. This involves substituting x with a virtual term t that does not include x and identifying the elimination set of ϕ, denoted as $elim^x(\phi)$. A virtual term itself is a sum $t + e\epsilon + z\mathbb{Z} + i\infty$. The new symbols $\epsilon, \mathbb{Z}, \infty$ do not occur in the result of applying virtual substitution. Instead, the virtual substitution function eliminates these auxiliary symbols. VIRAS shows how every literal in ϕ defines solution intervals with the lower bounds of these intervals being crucial for forming the elimination set $elim^x(\phi)$. To identify lower bounds of solution intervals, the paper expands on the properties of LIRA terms. The concepts of virtual terms, virtual substitutions, outer slope, bound distances, and segment slopes are defined to make the process of finding elimination sets easier. The difference between Cooper's method and VIRAS is that VIRAS extends the method to solve full LIRAS formulas. Cooper's method itself splits formulas into lower bounds, upper bounds and divisibility constraints, in comparison, VIRAS handles equivalence classes over $\mathbb{Z}$-terms to capture proper real numbers. The generalization for Cooper's method is not straightforward due to differences in bounds and equivalence classes over $\mathbb{R}$ and $\mathbb{Z}$. VIRAS does this by computing equivalence classes using *elim* over $\mathbb{Z}$-terms and core intervals, allowing for solutions that capture real numbers. VIRAS also introduces optimizations that enhance the efficiency of solving formulas, which are not present in Cooper's method.

A complementary approach to VIRAS comes with conflict- driven proof search for arithmetic reasoning, which is described in [Jovanović and de Moura, 2013]. The approach is limited to elimination sets with plain virtual terms, that is to virtual terms not containing ϵ or $\pm\infty$, but this is essential for VIRAS. VIRAS generalizes lemma learning from [Jovanović and de Moura, 2013], allowing to handle proper virtual terms

and improve VIRAS with conflict-driven proof search. The paper introduces CD-VIRAS by incorporating ϵ-lemmas, ∞-lemmas and $\mathbb{Z}$-flattening into VIRAS with conflict-driven proof search, resulting in a useful calculus. Adjusting two rules from the previous framework, $INNER\ CONFLICT$ and $LEAF\ CONFLICT$, the VIRAS calculus uses the lemma function $lemma_\phi$, which is defined to ensure soundness and completeness of the system. Soundness and completeness of the CD-VIRAS calculus is guaranteed. The function $lemma_\phi$ satisfies properties related to soundness and completeness, ensuring that the VIRAS calculus operates effectively and accurately in its proof search mechanism.

4.4.2 Fast Approximations of Quantifier Elimination

The paper [Garcia-Contreras et al., 2023] introduces a new quantifier reduction algorithm. As quantifier elimination is really costly, most of the time doubly exponential in the number of variables [Collins, 1976] or doubly exponential in the number of quantifier alternations, one can approximate it to get better performance for some specific cases. These approximations are called quantifier reductions. Note that when talking about quantifier reductions, there is a significant difference from full quantifier elimination. It might be possible that there are some free variables left after the reduction. The goal of [Garcia-Contreras et al., 2023] was to introduce a new quantifier reduction algorithm (QEL)

Traditional algorithms are implemented by a series of syntactic rules, operating directly on the syntax of an input formula. The key idea in this approach is to use the e-graph data-structure. QEL is based on the substitution rule $(\exists x\ x \approx t \wedge \varphi) \equiv \varphi[x \longmapsto t]$. With e-graphs, it is possible to eliminate multiple variables together, ensuring that a variable is eliminated if it is equivalent. E-graphs are data-structures to compactly represent a set of terms and an equivalence relation on those. Originally, e-graphs were proposed as a decision procedure for EUF. In QEL, e-graphs are used for quantifier reduction by selecting ground representatives. Nodes in an e-graph are labelled by function symbols or variables, and equivalence classes are formed using congruence closure. For more information on e-graphs, see [Willsey et al., 2021].

A big topic in [Garcia-Contreras et al., 2023] is the extraction of terms and formulas from e-graphs. To extract terms from e-graphs, a function called node-to-term is used. This function recursively constructs terms using a representative function that chooses one representative for each equivalence class. For formula extraction, term extraction is extended to full formulas by selecting representatives that maximize ground terms. The representative functions themselves are crucial for term extraction, as they contribute to the terms and variables that appear in the final reduction formula. By choosing representatives for ground nodes, ground terms are maximized. These nodes could be ground terms or could be built from ground terms.

The main contribution of [Garcia-Contreras et al., 2023] is the quantifier reduction algorithm QEL. This algorithm takes as input a formula φ with free variables v and outputs

a quantifier reduction of φ. The idea and the key steps of the algorithm can be described as follows:

1. Generating an e-graph from the input formula φ with free variables v.
2. Computing a representative function by finding ground definitions.
3. Finding additional non-ground definitions.
4. Finding a core to eliminate variables.
5. Produce a quantifier reduction from the generated e-graph.

The presented algorithm is relatively complete. This means that it eliminates a variable if it has a ground definition or can be represented by other variables.

Furthermore, the work [Garcia-Contreras et al., 2023] introduces Model-Based Projection (MBP) on top of QEL, to handle incomplete quantifier reductions. Specifically, MBP-rules for the theory of arrays and the theory of algebraic data-types are used to approximate quantifier elimination. MBP was first introduced for the SPACER CHC solver for LIA and LRA [Komuravelli et al., 2014]. And in the following extended to other theories. An MBP of a formula φ, with free variables v, relative to M is a quantifier-free formula ψ such that $\psi \Longrightarrow^{\exists}$ and M is a model of φ. To build an MBP-algorithm on top of QEL, algorithms to project variables that can not be eliminated cheaply are used. To implement the algorithm, model and theory specific projection rules are used. The presented algorithm is efficient and relatively complete, but does not guarantee to eliminate all variables.

The QEL and MBP-QEL approaches of [Garcia-Contreras et al., 2023] are implemented in Z3. The big key invention for these algorithms is that they work directly on the e-graph data structure, resulting in easier and faster procedures.

4.4.3 Playing with Quantified Satisfaction

The work presented in [Bjørner and Janota, 2015] introduces algorithms for satisfiability and quantifier elimination of quantified formulas. They showed how the algorithms handle theories that admit quantifier elimination, such as LIA, LRA and also other theories, for example, the theory of algebraic data-types. The presented algorithm itself is based on past progress in solving Quantified Boolean Formulas.

As a starting point, the algorithm Qesto [Janota and Marques-Silva, 2015] is generalized for non-propositional formulas. There exist several approaches for solving QBF formulas, for example, Lintao Zhang's scheme for combining DNF and CNF [Zhang, 2006], which was also extended to finite domains in [Bordeaux and Zhang, 2007], but there is no extension to infinite domains, which is needed for example, for LA. Qesto builds

on top of this scheme and also tracks dependencies between universally and existentially quantified variables and carefully encodes extra variables to capture these dependencies. The work of [Bjørner and Janota, 2015] came up with methods that, at the highest level, solve quantifier elimination as a game between two quantifiers and show how this could be implemented with a general algorithm. There are also other methods developed in QBF and SMT that connect with the presented approach, like [Goultiaeva et al., 2013, Goldberg and Manolios, 2014, Pugh, 1992, Janota et al., 2016].

The main contribution of [Bjørner and Janota, 2015] is QSAT, a general algorithm that views satisfiability of a quantified formula as a two-player game between existential and universal quantifiers. The algorithm alternates the moves at each quantifier level and then refines the constraints until the formula can be proven true or false. The algorithm takes a formula $G := \exists x_1 \forall x_2 \exists x_3 ... F$, which is closed and F is quantifier-free. Such a formula can be treated like a game between two players. The existential player, who tries to make the formula true by choosing values for $x_1, x_3, ...$ and the universal player, who tries to refute it by choosing values for $x_2, x_4, ...$ This is achieved by alternating between the formula F and its negation $\neg F$, which depends on the level of quantifier nesting. At odd levels, it works with F and at even levels with $\neg F$. With that idea, the authors set up a game-like structure for existential vs universal quantifiers. Further, all atomic subformulas are identified and tracked to come up with suitable models and also check the roles of each variable in the quantifier structure. Each atomic formula, or atom, is annotated with two levels: $level\forall(a)$, which is the highest index of any universally quantified variable appearing in atom a, and $level\exists(a)$, which is the highest index of any existentially quantified variable in atom a. The overall level of an atom is defined as the maximum of these two values. This should help the algorithm know how deep down an atom is in the quantifier hierarchy. Also, a function $level_j$ is defined, which adapts to the current quantifier level. If j is odd, then it uses the existential level of atoms and if j is even, the universal level. Furthermore, two helper functions are introduced. The first one is $strategy(M, j)$, which builds a conjunction of atoms that are determined by a previous model M, or to put it more simply, it encodes what the other player has already fixed in earlier moves. The other function is called $tail(j)$ and is used when applying quantifier elimination or model based projection. It refers to the sequence of quantifier blocks starting from the previous one and including the current and all later ones. The introduced algorithm QSAT starts at the outermost quantifier level and checks at each step if the formula at level F_j is satisfiable combined with the current *strategy*. If it is satisfiable, the model is updated and the algorithm proceeds to the next level by incrementing j. If not and $j = 1$, the formula is unsatisfiable, but if $j = 2$, the universal player has no winning move and the formula is satisfiable. Otherwise, the algorithm has to analyze the failure by extracting a minimal set of conflicting literals, compute a model-based projection to refine the strategy and jump back to an earlier level to retry with updated constraints. In the end, the loop is guaranteed to terminate, because each projection or refinement eliminates a possible model, and since there are only finitely many possible atoms and models, the search has to end eventually. Partial correctness

and termination of the algorithm is shown with the help of invariants.

The QSAT algorithm can be adapted to eliminate quantifiers. The key idea is just instead of deciding whether a formula is satisfiable, to construct an equivalent quantifier-free formula. Therefore, a modified version of QSAT is introduced. All in all, it works almost the same way as the described algorithm above, but it saves all the information learned from the solver. Each time the algorithm finds a conflict, it learns a little more about what the answer must exclude, and it also uses model-based projection to eliminate inner variables, so only free variables appear in the final answer. In the end, the learned constraints are combined in a way that eventually fully characterizes the input space, where the original formula is valid.

4.5 Further Works

There also exist other approaches for quantified reasoning. The focus of this thesis is on the three explained methods, but one also has to mention some other possibilities. In the following, some approaches are mentioned in more detail, while others are briefly mentioned.

4.5.1 Quantified Linear Arithmetic Satisfiability via Fine-Grained Strategy Improvement

In [Murphy and Kincaid, 2024] an approach for solving satisfiability of quantified LA formulas is introduced, which is similar to the approach in [Bjørner and Janota, 2015]. The improvement to previous approaches is based on game semantics. The used game semantics is based on FOL and gives meaning to a formula as a two-player game [Hintikka, 1982]. The paper overcomes some shortcomings of previous works by not only checking satisfiability but also synthesizing winning strategies for quantified satisfiability games. To give an overview, these games consider two players against each other, SAT and UNSAT, where SAT aims to prove satisfiability and UNSAT aims to disprove it. The goal of the games is not only to check satisfiability, but also to construct winning strategies applicable in tasks like invariant generation or program synthesis. The presented approach gives a recursive decision procedure that improves the strategy for both players incrementally, while avoiding transformations that can alter the game semantics. An example of such a transformation could be the prenex normal form. The focus of [Murphy and Kincaid, 2024] is on quantified LRA and LIA domains. The resulting decision procedure for checking satisfiability of quantified LA formulas develops a fine-grained structure of a formula to produce a winning strategy for SAT or UNSAT for both quantifiers and Boolean connectives.

The technique uses the fine-grained structure of LA formulas, which results in a recursive procedure that iteratively improves a candidate strategy via computing winning strategies to induce subgames. The fine-grained game semantics interpret formulas as

two-player games. Existential quantifiers (∃) and disjunctions (∨) are controlled by SAT, while universal quantifiers (∀) and conjunctions (∧) are controlled by UNSAT. If SAT has a winning strategy, the formula is satisfiable. This strategy is a mapping of moves, which ensures victory regardless of the response of UNSAT. A big point made by the authors is that fine-grained semantics maintain the structure of the original formula. With that, one avoids transformations and can introduce more quantifier alternations and , changing the meaning of the games. To capture multiple potential strategies, the concept of fine grained strategy skeletons is introduced. Skeletons are tree-like structures which map game states to possible moves. The fine grained strategy improvement algorithm uses these skeletons and refines them iteratively with counter-strategies, which should exploit weaknesses in the current strategy. This could be like the following scenario: if UNSAT can consistently counter the moves of SAT, the skeleton of SAT is adjusted to address these failures.

The main result of [Murphy and Kincaid, 2024] is the fine grained strategy improvement algorithm, which recursively refines strategies for quantified linear arithmetic games. In the initialization step, the algorithm starts with an initial skeleton for SAT. In the next step, UNSAT identifies counter strategies that exploit weaknesses in the skeleton of SAT. The algorithm solves subgames induced by these counter strategies and improves the skeleton of SAT if UNSAT fails to find a winning strategy. This whole process runs until one player has a definitive winning strategy. The presented iterative refinement avoids exhaustive quantifier elimination and is computationally efficient. The algorithm uses a model-based term selection to generate counter-strategies. Terms are selected based on if they satisfy quantifiers conditions in the current model. To approximate winning moves, for example for LRA, terms are used like bonds or symbolic averages. After the algorithm determines satisfiability, it constructs a winning strategy from the refined skeleton. Using constraint Horn clauses (CHCs), guards are computed for each move, ensuring they stick to the winning skeletons' rules. For practical applications, this step is really important, as it enables extracted strategies to inform tasks like program synthesis.

4.5.2 A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic

In [Rümmer, 2008], a constraint sequent calculus is introduced, designed to combine FOL and LIA. The presented calculus aims to address challenges in reasoning about quantified formulas that combine logical and arithmetic constraints. Essentially, it combines two existing approaches: free-variable tableaux with incremental closure [Giese, 2001] and the Omega quantifier elimination procedure [Pugh, 1991], which decides Presburger arithmetic (PA) [Presburger, 1929]. The paper [Rümmer, 2008] discusses the complexity of integrating FOL reasoning with theories, here LIA, and states that while most SMT solvers handle ground problems efficiently, support for quantified formulas remains limited. With the presented sequent calculus, [Rümmer, 2008] addressed this problem by enabling

systematic treatment of quantifiers alongside LA reasoning. The calculus itself is shown to be complete for function-free FOL with uninterpreted predicates, and it should also be able to decide the validity of PA formulas.

Firstly, a restricted calculus for pure FOL is introduced to illustrate how the framework of constrained sequents is related to normal free-variable tableau calculi. The introduced rules handle quantifiers and propositional connectives. To instantiate quantifiers, fresh constants are used to come up with free-variable reasoning. To ensure consistency, proofs propagate constraints generated by closing branches.

Following the restricted version, the calculus is extended to handle LIA. Therefore, new rules are introduced. For generating constraints by unifying complementary predicates with arithmetic terms, the pred-unify rule is introduced. Another rule needed is the close rule, which synthesizes constraints for proving sequents that involve linear inequalities and also divisibility. To ensure that all branches are closed with valid approximations, the constraints are passed through the proof tree. The presented calculus is sound and complete for fragments of FOL combined with the theory of LIA. In principle, this calculus is usable, but practically, it has a number of shortcomings. Therefore, built-in rules for handling LIA are defined that can be interleaved with the previous rules. That also yields a decision procedure for PA, which can be used to decide constraints. The rules for handling ground arithmetic constraints efficiently are: Omega Elimination, Fourier-Motzkin Elimination and Divisibility Rules. Omega Elimination replaces inequalities with simpler cases, which is essentially inspired by the Omega test for integer programming. Fourier Motzkin Elimination reduces complexity by handling variable eliminations for inequalities. Divisibility Rules convert divisibility constraints to equivalent arithmetic equations. With these rules for handling ground arithmetic constraints, performance could be improved. To ensure completeness, the concept of exhaustive proofs is introduced, where proofs are annotated with sets of universal constants. These sets represent the variables to eliminate. With this concept, the presented calculus guarantees that every provable formula is transformed into a closed, quantifier-free form, resulting in decision procedures for PA.

4.5.3 Ramsey Quantifiers in Linear Arithmetics

Another interesting approach is the work in [Bergsträßer et al., 2024]. For eliminating Ramsey quantifiers in LIA, LRA and also LIRA. With Ramsey quantifiers, one can assert the existence of cliques (complete graphs) in a graph induced by some formula. The approach works very well, but only for formulas with existential quantifiers. With such formulas, it runs in polynomial time and produces a formula of linear size. The algorithms of [Bergsträßer et al., 2024] lead to applications in proving termination/non-termination of programs, as well as checking variable dependencies (monadic decomposability) within a given formula. These results improve over older, more complex algorithms, which had triply exponential complexity. For evaluating the approach, a prototype was implemented for the Ramsey quantifier elimination algorithms for LIA, LRA, and LIRA in Python

using the Z3 interface Z3Py.

4.5.4 A Quantifier Elimination Algorithm for Linear Real Arithmetic

The paper [Monniaux, 2008] introduces a quantifier elimination algorithm for the theory of LRA. The presented approach works by combining SMT solving for model finding and also polyhedral projection for quantifier elimination, and it works with both existential and universal quantifiers. The SMT solver tests satisfiability of formulas with linear inequalities and returns models, and the projection $Project(C, v)$ eliminates variables v from the conjunction C using polyhedral projection. The introduced approach can be seen as an improvement over the approach of converting to DNF through ALL-SAT and performing projection, because it avoids exponential blow up. Compared to symbolic quantifier elimination like [Loos and Weispfenning, 1993], this approach is simpler to implement and more practical. Furthermore, the algorithm is also sound and complete.

4.5.5 Quantified Linear and Polynomial Arithmetic Satisfiability via Template-based Skolemization

In [Chatterjee et al., 2025], a new quantifier elimination method combining Skolemization for quantified LRA but also non-linear real arithmetic (NRA) is introduced. The approach is sound and also provides witnesses for existential quantifiers, but only semi-complete. The authors state that the complexity is in subexponential time and polynomial space. The following methods were introduced. A new method for quantifier elimination based on a novel template based Skolemization approach. Efficient satisfiability checking for LRA and NRA, which runs in subexponential time and polynomial space, parametrized by the size of Skolem function templates and is sound and semi-complete. Moreover, it also provides witnesses of satisfiability for the existentially quantified variables in the quantified formulas. The approach was also implemented as a prototype tool built with Python, which uses PolyHorn, Z3 and MathSAT5. The implementation showed really strong practical performance and a considerable improvement in the number of successful satisfiability checks, runtime and also unique satsfiability checks over Z3 and CVC5. The prototype is called QuantiSAT. All in all, the approach gives some advantages over other approaches, which have doubly-exponential time and at least exponential space complexity.

4.5.6 Yices-QS, an extension of Yices for quantified satisfiability

YicesQS, introduced in [Graham-Lengrand, 2021], is an extension of Yices 2, specifically designed to handle quantified formulas across several theories. It brings quantified reasoning into Yices 2 by integrating an advanced algorithmic framework called QSMA (Quantified Satisfiability Modulo Assignment) [Bonacina et al., 2023] and its optimized version, OptiQSMA. The QSMA algorithm formulates quantified solving as a kind of two-player game between $\forall$ and $\exists$ and generalizes the known and widely used CEGQI technique.

OptiQSMA further optimizes the algorithm with model-based over approximation and model-based under-approximation.

4.5.7 The Vampire Diary

The paper [Bártek et al., 2025] provides a historical and technical retrospective covering Vampire's evolution since 2013. It highlights technical innovations, design choices, implementation details, and community contributions that shaped Vampire into one of the most advanced first-order theorem provers based on the superposition calculus. The authors highlight that Vampire is not just a theorem prover but a living research platform that represents the progress of automated reasoning.

4.5.8 Quantifier Instantiations: To Mimic or To Revolt?

In [Jakubův and Janota, 2025], a novel instantiation approach that dynamically learns from existing instantiation techniques during solving is introduced. The authors want to adress the question if a new instantiation strategy should mimic existing successful ones or generate different new ones? The approach introduces a new probabilistic instantiation module (ProbGen) with the idea to treat existing instantiations (from other modules like CEGQI) as samples from a latent language. Further on, Probabilistic Context-Free Grammars (PCFGs) are used to model the generated language of instantiations. New candidate terms are generated using two modes: mimicking and revolt. With mimicking, terms are produced statistically similar to previously used ones, and with revolt probabilities are inverted to generate different terms. The authors found that probabilistic instantiation can boost solver performance beyond standard deterministic strategies, but mimicking alone is not enough, a balance between mimicking and revolt works best. Note that the method is complementary and does not replace existing techniques. The authors also conducted experiments in CVC5, which showed it consistently outperforms the baseline.

4.5.9 FMplex: A Novel Method for Solving Linear Real Arithmetic Problems

The authors introduce in [Nalbach et al., 2023] the idea of combining complementary solvers under one umbrella with a portfolio approach, which can leverage their strengths. With FMplex, they presented the first portfolio SMT solver dedicated to quantified LRA. Really interesting about this approach is that it uses several reasoning techniques: instantiation-based via CVC5, superposition-based via ALASCA/Vampire and quantifier elimination via Redlog [Dolzmann et al., 1999]. FMplex can be seen as a wrapper. It translates logical input into suitable formats for backend solvers and then runs each solver either in parallel or sequentially. After that, it collects answers and resolves inconsistencies, where FMplex relies on more complete solvers. The authors evaluated the approach with SMT-LIB quantified LRA examples against single solvers. The experiments showed that no solver dominates across all benchmarks, but a portfolio system like FMplex can

achieve better results overall. This also highlights the practical importance of hybrid approaches.

CHAPTER 5

Methodological Comparison

In the following, we take a closer look at the introduced approaches to compare them and give a complexity analysis, have a look at soundness and completeness, check practical use cases, summarize experimental results and provide an outlook for future work. We also state possible limitations.

5.1 Superposition-Based Methods

5.1.1 ALASCA: Reasoning in Quantified Linear Arithmetic

Complexity Analysis

The complexity of *ALASCA* [Korovin et al., 2023] is influenced by the handling of quantifiers, the integration of LA and of course the use of the superposition calculus. Quantified linear arithmetic with uninterpreted functions is Π^1_1-complete, which means that there is no sound and complete calculus in general. *ALASCA* tackles the challenge of being practical and does not focus in detail on theoretical complexity. Through unification with abstraction, variable elimination and theory-specific rules, the search space is reduced and unnecessary inferences are avoided. The performance itself depends on the structure of the input formulas and the number of quantifiers.

Soundness and Completeness

The inference rules of *ALASCA* are proven to be sound with respect to the partial axiomatization $A_{\mathbb{Q}}$ of LA. To put it simply, this means that any conclusion derived by *ALASCA* is valid under $A_{\mathbb{Q}}$. Further, unification with abstraction and the variable elimination rule is sound.

For ground clauses, *ALASCA* is complete with respect to $A_{\mathbb{Q}}$. Due to the Π^1_1-completeness,

ALASCA is not complete for quantified formulas in general. But *ALASCA* is refutational complete for sets of clauses without unshielded variables.

Practical Use Cases

The presented approach can be used in many real world scenarios: Program Verification, Formal Verification, Mathematical Reasoning, Financial Transactions and Web Security.

Experimental Results

The authors of [Korovin et al., 2023] state that the *ALASCA* approach outperforms state-of-the-art solvers using standard approaches like CVC5 and Vampire. The calculus itself is implemented in an extension of the Vampire theorem prover. Six sets of benchmarks, resulting in 6374 examples, are used. Benchmarks from the SMT-LIB repository [Barrett et al., 2016] are considered, which involve real arithmetic and uninterpreted functions. The sets are called LRA, NRA and UFLRA in SMT-LIB. Also, Sledgehammer examples generated by [Desharnais et al., 2022], which involve real arithmetic but do not use any other theories. This set is called SH. Further, two new sets of benchmarks are used, called TRIANGULAR and LIMIT, which contain reasoning challenges about triangular inequalities and continuous functions and problems that combine various limitation properties of real-valued functions. The benchmarks are compared with the solvers CVC5 [Barbosa et al., 2022], Vampire [Reger et al., 2022], Yices [Graham-Lengrand, 2022], UL-TELIM [Barth et al., 2022], SMTINT [Hoenicke and Schindler, 2022] and VERIT [Andreotti et al., 2022 The table of the results can be found in section 6 of [Korovin et al., 2023]. *ALASCA* achieves the overall best performance. It can be observed that *ALASCA* outperforms the two best arithmetic solvers of SMT-COMP 2022. The approach solves 118 more problems than CVC5, 159 more problems than Vampire and 213 more problems than Yices.

Limitations and Future Work

ALASCA currently focuses on LRA, extending it to handle LIA is a future direction to go. The authors discuss how more precise abstraction predicates could improve future proof search. Further, also improving literal/clause selections within *ALASCA* is a future topic.

5.1.2 Superposition Modulo Linear Arithmetic SUP(LA)

Complexity Analysis

In general, the combination of superposition calculus and LA reasoning in SUP(LA) [Althaus et al., 2009] is undecidable. The performance in practice depends on the structure of the input formulas and the number of quantifiers.

Soundness and Completeness

The inference rules of SUP(LA) are all sound with respect to LA. Further also the redundancy criteria ia sound and the LP-based constraint solving is sound.

For ground clauses, SUP(LA) is refutationally complete. In the quantified case, SUP(LA) is not complete. However, for sufficiently complete clause sets, the calculus is refutationally complete. The paper also introduces the notion of sufficient completeness, meaning that the calculus is complete for certain classes of problems.

Practical Use Cases

The presented approach can be used in many real world scenarios: Program Verification, Formal Verification, Mathematical Reasoning and Financial Transactions.

Experimental Results

The authors implemented the calculus in SPASS. It is shown that the implementation SPASS(LA) outperforms state-of-the-art solvers like Z3 on certain classes of problems. Note that the authors also mentioned that they did not do enough experiments to come up with a final conclusion, as the usage of solvers in SPASS(LA) differs from SMT scenarios. The experiments focus on transition systems and data structures. The comparison is only done with Z3, and it is easy to see that the new approach should outperform the basic solver.

Limitations and Future Work

SUP(LA) currently focuses only on LRA, an extension to handle also LIA would be a future direction. The approach can right now only be used for really specialized problems. Expanding the benchmark tests and testing SPASS(LA) on a broader range of problems should also be a future direction.

5.1.3 Beagle – A Hierarchic Superposition Theorem Prover

Complexity Analysis

As the combination of the hierarchic superposition and background theory reasoning in Beagle [Baumgartner et al., 2015] is undecidable in general, it is only possible to obtain refutational completeness for the calculus for sufficiently complete clause sets. In practice, the complexity is manageable for most cases, however performance depends on the structure of the input formulas and the number of quantifiers.

Soundness and Completeness

With respect to LIA and LRA, the inference rules of the hierarchic superposition calculus and the redundancy criteria are sound. Further, also the background theory solvers are

sound.

Beagle is in the ground case refutationally complete. Due to the Π^1_1-completeness, for quantified formulas, Beagle is not complete, but the calculus itself is refutationally complete for sufficiently complete clause sets. Sufficient completeness is ensured by adding definitions for BG-sorted terms, because this ensures that every ground BG-sorted FG term is equal to some BG term.

Practical Use Cases

Beagle can be used for: Program Verification, Formal Verification, Mathematical Reasoning and Financial Transactions.

Experimental Results

Beagle was tested on first-order problems from the TPTP-v6.1.0 problem library [Sutcliffe, 2009]. It tried 972 problems and solved 781 of them. Most interestingly, Beagle performed really well on the ARI (arithmetic) category, solving 531 out of 539 problems and also 41 out of 43 problems were solved in the number theory category. Beagle performs well on smaller problems with clear arithmetic constraints.

The prover was also tested on the 2014 release of the [SMT-LIB,] focusing on logics with arithmetics. Here, only problems indicated as unsatisfiable were selected. Beagle solved 89 problems that SMT solvers were unable to solve. Interestingly, Beagle struggled with problems that were marked trivial, and it also performed poorly on some quantifier-free problems. Note that many problems could not be parsed, and therefore, the conclusions are very vague.

Beagle also participated in the CASC-J7 competition [Sutcliffe, 2015] in the division of typed first-order arithmetic theorems. The prover solved 173 out of 200 problems and placed third in terms of overall problems solved. The winner of the competition was CVC4.

Limitations and Future Work

Beagle has some limitations, especially with large quantified problems and complex quantifier structures. Interestingly, the experiments also showed that Beagle struggles with a lot of trivial, quantifier-free problems. Therefore, improving integration with SMT solvers could be a future direction.

5.1.4 Comparison Table

	Strengths	Limitations
ALASCA: Reasoning in Quantified Linear Arithmetic [Korovin et al., 2023]	• Effective quantifier handling • Variable Elimination (VE) and Fourier-Motzkin (FM) rules • Replaces costly unification modulo theory with lazy abstraction • Integrated into Vampire • Complete and Sound • Effective for LRA + uninterpreted functions	• Completeness is restricted to $A_{\mathbb{Q}}$ • Requires specialized orderings and constraints for handling of non-ground terms • Limited to LRA
Superposition Modulo Linear Arithmetic SUP(LA) [Althaus et al., 2009]	• Refutationally complete for sufficiently complete clause sets • Decides satisfiability of first-order theories with universal quantification modulo LA • Applied to verify safety properties of transition systems	• Manual adjustments are needed for not sufficiently complete clause sets • Does not support combined theories • Current SPASS(LA) implementation lacks full equality reasoning
Beagle – A Hierarchic Superposition Theorem Prover [Baumgartner et al., 2015]	• Built-in solvers for LIA and LRA with optimizations • Introduces new simplification rules for theory reasoning • Implements calculus improvements like weak abstraction and determining (un)satisfiability w.r.t. quantification over finite integer domains • Automatically enforces sufficient completeness via the Define rule for certain classes of problems	• Struggles with large formulas • Simplification and certain strategies may break refutational completeness • Built-in rules are hard coded • No good performance on quantifier-free problems

Table 5.1: Comparison of [Korovin et al., 2023], [Althaus et al., 2009], [Baumgartner et al., 2015]

5.2 Instantiation-Based Methods

5.2.1 Solving Quantified Linear Arithmetic by Counterexample-Guided Instantiation

Complexity Analysis

The selection function for LRA, named S_{LRA} in [Reynolds et al., 2017] is inspired by the Loos-Weispfenning quantifier elimination method [Loos and Weispfenning, 1993]. This method has a worst case complexity of $O(2^{2^n})$ for formulas with n variables. The authors came up with a lazy instantiation strategy, which avoids enumerating all possible cases and therefore reduces the complexity. The selection function for LIA, named S_{LIA}, is related to Cooper's algorithm. The worst case complexity of Cooper's algorithm is doubly exponential. The function uses model-guided term selection to prioritize relevant instantiations, and therefore, the complexity is reduced in practice. For LIRA, the complexity is the same as quantifier elimination for LIRA, which is in general exponential in the number of variables. The worst-case complexity for counterexample-guided instantiation depends on the instantiation depth. The number of instantiations generated is lower than theoretical bounds, as experiments have shown.

Soundness and Completeness

The introduced procedure returns unsat if the formula is T-unsatisfiable and sat if T-satisfiable, respectively, and so the procedure is sound. The generated instance preserves the model properties and ensures that a satisfying model implies the model of the original formulas.

The authors of [Reynolds et al., 2017] prove that counterexample-guided instantiation is complete for one quantifier alternation in LRA and LIA. For multiple quantifier alternations, the completeness depends on the selection strategy. For arbitrary quantifier alternations, the procedure may not always terminate. All in all, the termination of the instantiation procedure is guaranteed if the selection function satisfies the finite and monotonic selection criteria.

Practical Use Cases

Counterexample-guided instantiation can be used for: Formal Verification and Program Synthesis.

Experimental Results

The procedure was implemented in the SMT solver CVC4 [Barrett et al., 2011]. The implementation was tested on quantified benchmarks over six classes in LRA and LIA of the SMT library. The 6 classes are: keaymaera, scholl, psyco, uauto, tpt and sygus. The experiments compare multiple configurations of CVC4 with state-of-the-art solvers. The

important configurations are: CVC4+LW (Loos-Weispfennig virtual term substitution), CVC4+FR (Ferrant-Rackoff with interior-point instantiation) and CVC4+NVT (Avoids virtual terms, substituting concrete values). Further, also the solvers Z3, Vampire, VeriT, Princess are used in the experiments. For the LRA benchmarks, the Yices solver was also used, and Beagle for LIA benchmarks.

The three benchmark classes for LRA were: keymaera, scholl and tpt. CVC4+LW solves 616 benchmarks out of 621 and therefore outperforms Z3 (615) and Yices (567). This is explainable because the approach does not require that quantified formulas must be in prenex normal form. Yices version 2.4.1 does not support nested quantification and therefore can not be used in the scholl category. CVC4+LW solved 100% in the scholl category, which tests the ability to handle alternating quantifiers and Boolean combinations. CVC4 also solves the same instances as CVC4+LW, but is slightly slower. The best LRA automated theorem prover, Vampire, solves only 347 benchmarks, compared to 616 solved by CVC4+FR

For the LIA benchmarks the classes were: psyco, sygus, tptp and uauto. CVC4 and Z3 solved all 461 benchmarks. The best LIA automated theorem prover was also Vampire, which solves only 284 benchmarks and it scored 100% only in the class uauto.

Limitations and Future Work

The authors of [Reynolds et al., 2017] introduced an approach for LIRA, but completeness has not yet been proven. This would be a topic for future work. Also, in some cases, deeply nested formulas may require more instantiations. In the future, new theories could also be added and the authors also want to focus on further heuristics for quantified LA with arbitrary quantifier alternations. For LIA, large coefficients can lead to many instantiations, this should also be looked at for future work. The long term goal of the authors would be to develop an approach that is effective in practice for quantified formulas involving background theories and also uninterpreted functions.

5.2.2 Theory Instantiation

Complexity Analysis

In general, complexity of the theory reasoning backend depends on the efficiency of the SMT solver or ground reasoner for the given background theory T. For instantiation in [Ganzinger and Korovin, 2006], the process can involve many instantiations, which can cause a combinatorial explosion in worst case. The fair saturation process either stops after a finite number of steps, detecting satisfiability or unsatisfiability of the initial clause set, or if the limit of a saturated set is obtained, and hence the initial clause set is satisfiable.

Soundness and Completeness

Because of the black-box style integration of theory reasoning and by grounding literals before instantiation, the introduced framework is sound. That means that any derivation of unsatisfiability corresponds to actual unsatisfiability modulo the theory T.

In order to achieve refutational completeness, the theory reasoner must be answer-complete for Unit Calculus and also complete for ground clauses. In the paper, a theorem establishes that if the saturation process is $TInst - fair$ then it is complete. That means it does not ignore valid inference opportunities.

Practical Use Cases

Theory instantiation can be used for: SMT-solving and also Verification of Software and Hardware.

Experimental Results

In the paper [Ganzinger and Korovin, 2006], the approach was not implemented yet, but moving forward, there is an implementation of the approach in iProver, and there are also some experiments [Korovin, 2008].

The authors of [Ganzinger and Korovin, 2006] evaluated the approach on the standard benchmark for first-order theorem provers – TPTP library v3.2.0. In the FOF division, iProver was in the top three provers and also in the EPR division, iProver was very good.

Limitations and Future Work

There are some limitations to the approach of [Ganzinger and Korovin, 2006]. First of all, not all theory reasoners satisfy answer completeness, which is needed. Further, the framework is general, so there is no theory specification. For certain specific theories, the approach might be less efficient. For future work, the authors would like to extend the approach to theories with particular properties, like Shostak theories [Ganzinger et al., 2003]. This is essential for integrating reasoning with fragments of arithmetic. Also, the authors would like to study the relationship between the presented approach and hierarchical reasoning [Bachmair et al., 1994].

5.2.3 Syntax-Guided Quantifier Instantiation

Complexity Analysis

The approach of [Niemetz et al., 2021] integrates syntax-guided synthesis into quantifier instantiation, which involves enumerating terms from a grammar. The grammar size influences the complexity. Larger grammars increase the search space, but smart enumeration with symmetry breaking makes it less complex. The number of terms generated depends on the expressiveness of the grammar. Each iteration of syntax-guided quantifier

instantiation (SyQI) performs two SAT/SMT checks. These checks are computational expensive. Further, the cost of generating lemmas depends on the size of synthesized terms and the theory reasoning which is required. The paper itself does not provide explicit asymptotic bounds. But it is worth mentioning that similar approaches have theoretical exponential complexity in the worst case, and SyQI is competitive with other methods.

Soundness and Completeness

SyQI has refutational soundness, as if it returns unsat, the input formula is T-unsatisfiable. This holds because the added lemmas are entailed by the input. SyQI has also model soundness, as if SyQI returns sat, the input formula is T-satisfiable. This could be explained by the fact that no counterexample exists, so the model satisfies all universal formulas.

Progress for SyQI is ensured by adding new lemmas in each iteration, thereby preventing infinite loops. The authors of [Niemetz et al., 2021] show that SyQI is refutationally complete if the provided grammars are complete. This holds for theories admitting quantifier elimination, like LA. For theories without quantifier elimination, completeness is not guaranteed. SyQI is not guaranteed to terminate for satisfiable formulas, but it is sometimes possible to find models when grammars include sufficient terms.

Practical Use Cases

SyQI can be used for: Software Verification, Automated Theorem Proving and Synthesis.

Experimental Results

SyQI is implemented in the SMT solver CVC4 [Barrett et al., 2011]. The approach was evaluated on all configurations and on all quantified logics in [SMT-LIB,] that do not contain uninterpreted functions. Logics with uninterpreted functions are excluded, as such logics are not expected to be more efficient than heuristic instantiation techniques, such as E-matching. There are a lot of logics included, for the scope of this thesis, most importantly LIA and LRA. The benchmark set consists of 15746 benchmarks.

The first experiment was to determine the best combination of scope-based and size-based ground term selection strategies for grammar construction. Here, the strategy *both* for the scope performs best, and all size-based strategies perform equally well.

The second experiment was to determine the best lemma selection strategy out of the three strategies *priority-inst*, *priority-eval* and *interleave*. The best strategy overall was *interleave*. Prioritizing evaluation lemmas over instantiation lemmas performed worse than the other configurations.

Finally, the new approach SyGuS was compared to other techniques implemented in CVC4, the state-of-the-art SMT solvers Z3, Boolector, and the superposition-based theorem prover Vampire. Boolector implements counterexample-guided model synthesis, but only supports the logic BV, while Vampire supports LIA, LRA, NIA and NRA. The following configurations of CVC4 are used: ematch (with E-matching enabled), CEGQI (with LA and bit vectors enabled), enum (with enumerative instantiation enabled) and syqi (with the presented SyGuS instantiation approach enabled). For term selection, the strategy both-both is used, and for lemma selection, interleave is used. E-matching performs very poorly on the benchmark sets, which was more or less expected, as it is designed with a focus on problems with uninterpreted functions. Enumerative instantiation also performs poorly, this could be because it is not designed for inputs without uninterpreted functions. SyQI solves 556 more benchmarks than enumerative instantiation. This enhances the need for a syntax-guided approach to instantiation for inputs rich in background theories. SyQI is also competitive compared to CEGQI, which uses the best known instantiation strategies. Interestingly, counterexample-guided instantiation outperforms SyQI on logics such as LIA and LRA. For LRA, syntax-guided techniques are ineffective, since it is often important to construct specific real constants based on solving sets of linear equalities and inequalities.

CVC4 with counterexample-guided instantiation outperforms Z3 and Vampire in LIA, but for example in LRA, Z3 performs best. To summarise, theory-specific approaches outperform SyQI in categories such as LIA and LRA. But it is also worth mentioning that the presented approach performs very well on quantified floating-point arithmetic.

Limitations and Future Work

As we have seen before, SyQI underperforms against theory-specific methods in logics like LIA and LRA. Further, completeness relies on the grammar's ability to generate all necessary terms and also termination is not guaranteed for satisfiable formulas in undecidable theories. For future work, an automated grammar refinement would be a good way to go. Further, it would be nice to provide an interface that allows users to use their own grammars in SyQI. Also, the authors want to use their approach as a baseline for quantified logics in recent and new theories.

5.2.4 Comparison Table

	Strengths	Limitations
Solving Quantified Linear Arithmetic by Counterexample-Guided Instantiation [Reynolds et al., 2017]	• Sound and complete for LRA and LIA with one quantifier alternation • Non-Prenex Handling • Integrates easily with SMT solver architectures • Outperforms state-of-the-art solvers	• Incomplete for LIRA • Requires finite and monotonic selection functions • Performance may slow down for formulas with many variables due to exponential growth in possible instantiations
Theory Instantiation [Ganzinger and Korovin, 2006]	• Black-box integration of theories, only a theory reasoner that is complete on ground clauses and answer-complete on unit clauses is needed • Combination of different theories possible • Guaranteed completeness	• Underlying Unit Calculus must be answer complete • Ground solver must be fully complete on finite ground sets • Deep quantifier alternations are expensive • Only works for theories specified as universal axioms
Syntax-Guided Quantifier Instantiation [Niemetz et al., 2021]	• Works across multiple theories and combinations without requiring theory-specific instantiation rules • Iteratively refines models using synthesized terms • Reduces redundancy by avoiding equivalent term variants • Effective for quantified floating-point arithmetic	• Underperforms for theory-specific methods • Completeness is based on grammar quality • No guarantee of termination for satisfiable formulas in undecidable theories • No possibility for user-provided grammars

Table 5.2: Comparison of [Reynolds et al., 2017], [Ganzinger and Korovin, 2006], [Niemetz et al., 2021]

5.3 Quantifier Elimination in Computer Algebra

5.3.1 VIRAS: Conflict-Driven Quantifier Elimination for Integer-Real Arithmetic

Complexity Analysis

First of all, it is to mention that the authors of [Schoisswohl et al., 2024] state that finding actual complexity bounds for the presented method remains for future research. However, there could be some characteristics of complexity. VIRAS avoids exponential blow-up by overcoming the trouble of arithmetic normalizations. The authors showed in an example that VIRAS reduces the elimination set size to $O(n^2)$ instead of $O(3^n)$, by not using external quantifier elimination procedures but operating directly on LIRA terms. Conflict-driven VIRAS reduces the search space by avoiding redundant assignments, thereby reducing complexity.

Soundness and Completeness

The VIRAS method is proven to be sound, meaning that if the quantifier elimination procedure comes up with a quantifier-free formula equivalent to the original quantified formula, then this equivalence holds in the theory of LIRA. The construction of elimination sets and the virtual substitution function are designed in a way that no solutions are missed and no wrong solutions are introduced, which ensures soundness.

VIRAS is also complete, meaning that if the original quantified formulas have an existing truth assignment, then also the quantifier-free formula resulting from VIRAS will be true.

Further, also the conflict-driven proof search CD-VIRAS is both sound and complete.

Practical Use Cases

VIRAS could be used for: Formal Verification of Software and Hardware and Security Analysis.

Experimental Results

At the time of writing this thesis, VIRAS has not been evaluated as the implementation is still ongoing.

Limitations and Future Work

The VIRAS method relies on intricate virtual substitutions and, therefore, an implementation is complicated. VIRAS supports arbitrary quantifier alternations, but deeply nested quantifiers could require recursive quantifier elimination steps, which increases the complexity. For future work, computing tighter bounds $\text{distY}^{\pm}$ is a topic, and also

computing more accurate discontinuity sets. Also, implementing VIRAS in the theorem prover Vampire is a challenge for future work.

5.3.2 Fast Approximations of Quantifier Elimination

Complexity Analysis

Traditional quantifier elimination is computationally expensive, as its doubly exponential. The quantifier reduction method of [Garcia-Contreras et al., 2023] avoids full elimination by reducing the number of variables using an efficient e-graph-based method. The approximation is sufficient in most cases and is not as computationally expensive.

Soundness and Completeness

The introduced quantifier elimination approximation QEL and MB-QEL are semantically sound, because every transformation preserves logical entailment.

QEL is relatively complete with respect to some semantic properties of the input formula. If a variable is equal to a ground term, QEL will eliminate it. QEL is unaffected by variable orderings and syntactic rewrites, since it is relatively complete. MBP-QEL is not complete, but provides under-approximations of QEL by a model.

Practical Use Cases

QEL/MBP-QEL could be used for: SMT-Solving, Model-Checking, CHC-Solving or Program Synthesis and Formal Verification.

Experimental Results

QEL and MBP-QEL were implemented in Z3. The implementation itself is referred to as Z3EG and evaluated using two tasks.

The first evaluation is on the QSAT algorithm, which is also discussed in [Bjørner and Janota, 2015], for checking satisfiability of formulas with alternating quantifiers. Three QSAT implementations are compared: the existing version in Z3, Z3EG and the QSAT implementation in YicesQS. Benchmarks are used in the theory of quantified LIA and LRA from SMT-LIB [Barrett et al., 2016], with alternating quantifiers. Further, also two modified variants of the LIA and LRA benchmarks are used, where some non-recursive ADT variables are added. Considering LIA, Z3EG, and Z3 solve all benchmarks in under a minute, while YicesQS is unable to solve all instances. For LRA, YicesQS solves all instances, Z3 only some and Z3EG performs similarly to Z3. In LRA, the new algorithms are not being used as there are not many equalities in the formula, and no equalities are inferred during the run of QSAT. For the mixed ADT and arithmetic, YicesQS could not be used as it does not support ADTs. The benchmarks show that Z3EG solves many more instances than Z3.

The second evaluation shows the efficacy of MBP-QEL for arrays and ADTs for CHC-solving inside the SPACER solver. Here, Z3 and Z3EG are compared with ELDARICA [Hojjat and Rümmer, 2018]. Two sets of benchmarks are used to test MBP-QEL. The first set is for the verification of Solidity smart contracts [Alt et al., 2022], which nest ADTs and arrays and is suitable to test MBP-QEL. Z3EG solves more instances than Z3, and it could be observed that MBP-QEL makes a significant impact on SPACER. Interestingly, Z3EG also solves almost as many instances as ELDARICA. The second set is from the Array benchmarks from the CHC competition [Gurfinkel et al., 2018]. Z3EG solves two additional safe instances and almost as many unsafe instances as Z3, and both Z3EG and Z3 solve more instances than ELEDARICA.

Limitations and Future Work

In general, QEL is not a full quantifier elimination method as it is only an approximation. MBP depends on the guiding model, and the quality of the result may vary. QEL may produce some redundancies in the final formula. The approach requires integration into e-graph-based solvers, otherwise, it will not be usable. For future work, QEL could be extended with more theories. Also MBP rules could be optimized to work better across combined theories. Another topic could be to explore smarter variable elimination orders and representations.

5.3.3 Playing with Quantified Satisfaction

Complexity Analysis

The authors of [Bjørner and Janota, 2015] did not explicitly give a formal complexity bound for the QSAT algorithm, but based on the design, it is possible to give a theoretical complexity. The complexity of the algorithm increases exponentially with the number of quantifier alternations. For the propositional case, QBF is PSPACE-complete, but extended to LRA or LIA, the complexity is becomes even higher. The key subroutine is model-based projection, which depends on the theory. For LRA, quantifier elimination via model-based projection can be doubly exponential due to quantifier alternations. For LIA, elimination procedures can be triple exponential in worst case. Each time the algorithm refines a strategy, it adds a new constraint to the formula, the number of refinements is bounded by the number of distinct projections, which in LRA and LIA can be exponential or more.

Soundness and Completeness

The introduced QSAT algorithm is sound, as it is shown that whenever it terminates, it returns the correct truth-value of the quantified input formula. This is established by a theorem, which states that the algorithm is partially correct, which means that when it terminates, it correctly determines if the input formula is true or false. This also applies to the variant of the QSAT algorithm that constructs a quantifier-free formula. Although there is no theorem given, the correctness proof can be established by induction on the

quantifier depth.

QSAT is not guaranteed to terminate on all theories, but under LIA and LRA, all conditions for termination are fulfilled, and the algorithm terminates. This is achieved under a special requirement, which states that each model-based projection operation must produce a finite set of possible projections and must cover the quantified elimination. Therefore, QSAT is complete for LIA and LRA. This also holds for the variant of QSAT for quantifier elimination.

Practical Use Cases

The introduced approach can be used for: Formal Verification of Software and Hardware and Program Synthesis.

Experimental Results

The QSAT algorithm is evaluated against the algorithm QT [Phan et al., 2012]. All in all, the new QSAT algorithm is an improvement over QT. The algorithms were evaluated using a benchmark from [Phan et al., 2012]. The task was to solve 64 quantified LIA benchmarks. Here, QSAT solves all benchmarks almost instantaneously, while QT has 10 timeouts and takes much longer for the benchmarks. Further, the algorithms were also evaluated using benchmarks from the SMT-LIB2 suite for LRA. The authors of [Bjørner and Janota, 2015] mention that many of the benchmarks have been randomly generated and that this is reflected in overall fluctuations in the number of Simplex pivoting steps taken to check satisfiability. But the QSAT algorithm performs much better overall, as QT times out for 42 benchmarks, QSAT only times out for two benchmarks.

Limitations and Future Work

The approach of [Bjørner and Janota, 2015] relies heavily on the ability to compute model-based projection to eliminate quantifiers, which could be seen as a limitation, as not all theories support model-based projection (note that LIA and LRA do) and also, the projection may generate infinitely many distinct projections. The approach is also very sensitive to quantifier alternations, as performance may drop with deeply nested quantifiers. Also, there is no guarantee for minimality in quantifier elimination. The authors state a lot of future work directions. They would like to extend the algorithms to use more powerful strategies that can be very useful in the Boolean case. They also want to extend this approach to other theories for model-based projection, and also state when and how one can combine theories with projection. Further, the approach should also be extended to solve reachability games.

5.3.4 Comparison Table

	Strengths	Limitations
VIRAS: Conflict-Driven Quantifier Elimination for Integer-Real Arithmetic [Niemetz et al., 2021]	• Handles arbitrary quantifier alternations over LIRA • Combines Cooper's method for LIA-QE and virtual substitution for full LIRA support and avoids exponential blowup • Works directly on original LIRA terms and so its elimination sets grow only polynomially in many cases	• Finding actual complexity bounds is challenging • Complex implementation effort
Fast Approximations of Quantifier Elimination [Garcia-Contreras et al., 2023]	• QEL eliminates any variable that is provably equal to a ground term in the original formula due to the e-graph data-structure • Good in handling of implicit equalities • Integrated model-based projection for combined theories	• QEL/MBP-QEL may leave some variables uneliminated when there is no ground or congruence-derived definition • Extraction depends on finding an admissible representative function • Overall precision of MBP-QEL is only as good as the underlying rewrite rules
Playing with Quantified Satisfaction [Bjørner and Janota, 2015]	• Works for propositional QBF and also first-order theories that admit quantifier elimination (LIA, LRA) • Two-player game alternating quantifiers, which reduces wasted search • Sound and, under model-based projection assumptions, complete	• Completeness and termination relies on background theory providing a finite model-based projection • Deep quantifier alternations still have exponential blowup • Quantifier elimination output could be very large

Table 5.3: Comparison of [Schoisswohl et al., 2024], [Garcia-Contreras et al., 2023], [Bjørner and Janota, 2015]

CHAPTER 6

Benchmarks and Experiments

6.1 Configuration

To compare the various approaches, we implemented our own benchmarks on our own notebook. All benchmarks and experiments were conducted on a Lenovo ThinkPad E495 laptop equipped with an AMD Ryzen 5 3500U quad-core processor, 8 GB of DDR4 RAM, a 256 GB NVMe SSD, and integrated AMD Radeon Vega 8 graphics. The benchmarks are from the 20th International Satisfiability Modulo Theories Competition (SMT-COMP 2025) [Preiner et al., 2025]. The dataset can be found here: `https://zenodo.org/records/15493090`. For the evaluation, the non-incremental benchmarks for LIA and LRA were used. They are designed so that each benchmark is a self-contained problem, and the solver is expected to process each one from scratch, without relying on any state or learned information from previous problems. We use it in that way so we can focus on the core reasoning capabilities. For the benchmarks, 100 examples were randomly selected from LIA and LRA. The selected files are as follows:

```
LRA/scholl-smt08/RNDPRE/RNDPRE_3_22.smt2
LRA/2010-Monniaux-QE/mjollnir3/formula_167.smt2
LRA/scholl-smt08/RNDPRE/RNDPRE_4_36.smt2
LRA/2010-Monniaux-QE/mjollnir1/formula_027.smt2
LRA/2010-Monniaux-QE/mjollnir5/formula_271.smt2
LRA/scholl-smt08/RNDPRE/RNDPRE_4_52.smt2
LRA/2010-Monniaux-QE/mjollnir5/formula_050.smt2
LRA/2010-Monniaux-QE/mjollnir3/formula_092.smt2
LRA/2010-Monniaux-QE/mjollnir5/formula_276.smt2
LRA/2010-Monniaux-QE/mjollnir1/formula_277.smt2
LRA/2010-Monniaux-QE/mjollnir5/formula_150.smt2
LIA/tptp/ARI572=1.smt2
LRA/2010-Monniaux-QE/mjollnir5/formula_069.smt2
LIA/UltimateAutomizer/recHanoi03_true-unreach-call_true-termination.c_847.smt2
LIA/20250213-Frobenius/fcp_269_271_277.smt2
LIA/UltimateAutomizer/nested9_true-unreach-call.i_755.smt2
LRA/keymaera/water_tank-node31000.smt2
LRA/2010-Monniaux-QE/mjollnir3/formula_166.smt2
LRA/2010-Monniaux-QE/mjollnir1/formula_182.smt2
LRA/scholl-smt08/RNDPRE/RNDPRE_3_21.smt2
LRA/2010-Monniaux-QE/mjollnir1/formula_006.smt2
LIA/UltimateAutomizer/nested9_true-unreach-call.i_775.smt2
```

```
LRA/scholl-smt08/RNDPRE/RNDPRE_4_33.smt2
LRA/2010-Monniaux-QE/mjollnir6/formula_191.smt2
LRA/2010-Monniaux-QE/mjollnir2/formula_178.smt2
LRA/2010-Monniaux-QE/mjollnir6/formula_028.smt2
LIA/psyco/049.smt2
LIA/tptp/ARI032=1.smt2
LRA/2010-Monniaux-QE/mjollnir5/formula_038.smt2
LRA/2010-Monniaux-QE/mjollnir2/formula_135.smt2
LRA/scholl-smt08/RNDPRE/RNDPRE_4_26.smt2
LRA/2010-Monniaux-QE/mjollnir2/formula_201.smt2
LRA/scholl-smt08/model/model_5_26.smt2
LIA/psyco/034.smt2
LIA/tptp/ARI045=1.smt2
LRA/2010-Monniaux-QE/mjollnir4/formula_137.smt2
LRA/2010-Monniaux-QE/mjollnir5/formula_121.smt2
LRA/2010-Monniaux-QE/mjollnir3/formula_178.smt2
LRA/2010-Monniaux-QE/mjollnir1/formula_132.smt2
LRA/scholl-smt08/RND/RND_4_19.smt2
LRA/keymaera/intersection-example-simple.proof-node431299.smt2
LIA/20250213-Frobenius/fcp_179_181_191.smt2
LRA/2010-Monniaux-QE/mjollnir5/formula_022.smt2
LRA/scholl-smt08/RNDPRE/RNDPRE_3_13.smt2
LRA/keymaera/intersection-example-simple.proof-node44393.smt2
LRA/2010-Monniaux-QE/mjollnir4/formula_139.smt2
LRA/2010-Monniaux-QE/mjollnir1/formula_238.smt2
LRA/2010-Monniaux-QE/mjollnir3/formula_048.smt2
LRA/2010-Monniaux-QE/mjollnir6/formula_141.smt2
LIA/psyco/158.smt2
LRA/2010-Monniaux-QE/mjollnir2/formula_115.smt2
LIA/20250213-Frobenius/fcp_5_7_11.smt2
LIA/20250213-Frobenius/fcp_167_173.smt2
LRA/keymaera/intersection-example-simple.proof-node409496.smt2
LRA/2010-Monniaux-QE/mjollnir4/formula_114.smt2
LRA/scholl-smt08/RNDPRE/RNDPRE_4_55.smt2
LRA/2010-Monniaux-QE/mjollnir2/formula_040.smt2
LRA/keymaera/water_tank-node5020.smt2
LRA/2010-Monniaux-QE/mjollnir5/formula_254.smt2
LRA/2010-Monniaux-QE/mjollnir3/formula_220.smt2
LRA/2010-Monniaux-QE/mjollnir1/formula_155.smt2
LIA/UltimateAutomizer/recHanoi03_true-unreach-call_true-termination.c_1757.smt2
LRA/2010-Monniaux-QE/mjollnir4/formula_187.smt2
LRA/2010-Monniaux-QE/mjollnir1/formula_017.smt2
LRA/2010-Monniaux-QE/mjollnir1/formula_185.smt2
LRA/scholl-smt08/RNDPRE/RNDPRE_4_3.smt2
LRA/2010-Monniaux-QE/mjollnir6/formula_086.smt2
LRA/2010-Monniaux-QE/mjollnir2/formula_274.smt2
LIA/psyco/018.smt2
LRA/2010-Monniaux-QE/mjollnir1/formula_206.smt2
LRA/scholl-smt08/RND/RND_3_18.smt2
LRA/2010-Monniaux-QE/mjollnir6/formula_060.smt2
LRA/2010-Monniaux-QE/mjollnir2/formula_233.smt2
LIA/psyco/154.smt2
LRA/2010-Monniaux-QE/mjollnir2/formula_035.smt2
LRA/2010-Monniaux-QE/mjollnir1/formula_255.smt2
LRA/2010-Monniaux-QE/mjollnir6/formula_192.smt2
LRA/2010-Monniaux-QE/mjollnir3/formula_245.smt2
LRA/2010-Monniaux-QE/mjollnir4/formula_050.smt2
LIA/psyco/037.smt2
LRA/2010-Monniaux-QE/mjollnir4/formula_012.smt2
LIA/psyco/160.smt2
LRA/2010-Monniaux-QE/mjollnir5/formula_186.smt2
LRA/scholl-smt08/RND/RND_4_18.smt2
LRA/keymaera/water_tank-node16055.smt2
LRA/2010-Monniaux-QE/mjollnir1/formula_114.smt2
LRA/2010-Monniaux-QE/mjollnir2/formula_254.smt2
LRA/2010-Monniaux-QE/mjollnir3/formula_234.smt2
LRA/scholl-smt08/model/model_6_37.smt2
LIA/UltimateAutomizer/Primes_true-unreach-call.c_127.smt2
LRA/2010-Monniaux-QE/mjollnir2/formula_241.smt2
LRA/keymaera/intersection-example-simple.proof-node404096.smt2
LIA/psyco/039.smt2
LIA/20190429-UltimateAutomizerSvcomp2019/jain_2_true-unreach-call_true-no-overflow_false-
    termination.i_12.smt2
```

```
LRA/2010-Monniaux-QE/mjollnir2/formula_231.smt2
LIA/UltimateAutomizer/nested9_true-unreach-call.i_927.smt2
LRA/scholl-smt08/RND/RND_3_10.smt2
LIA/UltimateAutomizer/recHanoi03_true-unreach-call_true-termination.c_287.smt2
LRA/2010-Monniaux-QE/mjollnir2/formula_186.smt2
LRA/2010-Monniaux-QE/mjollnir1/formula_107.smt2
```

For the benchmarks, the following program was constructed:

```python
import os
import subprocess
import csv
import time

# Configuration
SOLVER = ""
SOLVER_OPTS = [""]
TIMEOUT =
ROOT_DIR = ""
RESULTS_DIR = ""

def main():
    summary_rows = []

    with open("selected_benchmarks.txt") as f:
        benchmark_files = [line.strip() for line in f if line.strip()]

    for file_path in benchmark_files:
        if not file_path.endswith(".smt2"):
            continue
        file_path = ROOT_DIR + "/" + file_path
        os.makedirs(RESULTS_DIR, exist_ok=True)
        base = os.path.splitext(os.path.basename(file_path))[0]
        out_file = os.path.join(RESULTS_DIR, f"{base}.out")
        metrics_file = os.path.join(RESULTS_DIR, f"{base}.metrics")

        print(f"Running solver on {file_path} ...")
        start_time = time.time()

        # Build command: /usr/bin/time -v timeout TIMEOUT SOLVER SOLVER_OPTS file_path
        cmd = ["/usr/bin/time", "-v", "timeout", str(TIMEOUT), SOLVER] + SOLVER_OPTS + [file_path]


        with open(out_file, "w") as out, open(metrics_file, "w") as metrics:
            proc = subprocess.run(cmd, stdout=out, stderr=metrics)

        end_time = time.time()

        result = None
        with open(out_file) as f:
            for line in f:
                line = line.strip()
                if line.lower() in ("sat", "unsat", "unknown"):
                    result = line
                    break

        wall_time = user_time = sys_time = max_mem = None
        with open(metrics_file) as f:
            for line in f:
                if "Elapsed (wall clock) time" in line:
                    wall_time = line.split(":", 1)[1].strip()
                elif "User time (seconds)" in line:
                    user_time = line.split(":", 1)[1].strip()
                elif "System time (seconds)" in line:
                    sys_time = line.split(":", 1)[1].strip()
```

```
                elif "Maximum resident set size" in line:
                    max_mem = line.split(":", 1)[1].strip()

        summary_rows.append({
            "file": file_path,
            "result": result if result else "no_result",
            "exit_code": proc.returncode,
            "wall_time": wall_time,
            "user_time": user_time,
            "sys_time": sys_time,
            "max_mem_kb": max_mem,
            "start_time_epoch": start_time,
            "end_time_epoch": end_time,
        })

    os.makedirs(RESULTS_DIR, exist_ok=True)
    summary_csv = os.path.join(RESULTS_DIR, "summary.csv")
    with open(summary_csv, "w", newline="") as csvfile:
        fieldnames = ["file", "result", "exit_code", "wall_time", "user_time", "sys_time", "
     max_mem_kb", "start_time_epoch", "end_time_epoch"]
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
        writer.writeheader()
        for row in summary_rows:
            writer.writerow(row)

    print(f"\nBenchmarking complete. Summary written to {summary_csv}")

if __name__ == "__main__":
    main()
```

Listing 6.1: Benchmark python code

6.2 Benchmarking

For benchmarking the approach of [Korovin et al., 2023], the implementation from `https://github.com/vprover/vampire/tree/alasca` was used. The configuration was as follows:

```
SOLVER = "vampire"
SOLVER_OPTS = ["--mode", "portfolio", "--output_mode", "smtcomp", "-alasca", "on", "-uwa", "
     alasca_main", "-to", "qkbo"]
TIMEOUT = 61
ROOT_DIR = "non-incremental"
RESULTS_DIR = "results_ALASCA"
```

Listing 6.2: Configuration for ALASCA

The approach of [Althaus et al., 2009] was implemented in the SPASS theorem prover, but the implementation itself is not public, and therefore, benchmarks are not possible.

The approach of [Baumgartner et al., 2015] was implemented in the Beagle theorem prover, which can be found here: `https://bitbucket.org/peba123/beagle/src/master/`. The configuration was as follows:

```
SOLVER = "beagle"
SOLVER_OPTS = []
TIMEOUT = 61
```

```
ROOT_DIR = "non-incremental"
RESULTS_DIR = "results_BEAGLE"
```

Listing 6.3: Configuration for Beagle

The approach of [Ganzinger and Korovin, 2006] was implemented in the iProver theorem prover, which can be found here: `https://gitlab.com/korovin/iprover`. The configuration was as follows:

```
SOLVER = "./iproveropt-multi-core.sh"
SOLVER_OPTS = []
TIMEOUT = 61
ROOT_DIR = "non-incremental"
RESULTS_DIR = "results_iProver"
```

Listing 6.4: Configuration for iProver

The implementation of the approach of [Reynolds et al., 2017] was implemented in CVC4, can be found here: `https://github.com/pysmt/CVC4` and is tested with the following configuration:

```
SOLVER = "cvc4"
SOLVER_OPTS = ["--cegqi"]
TIMEOUT = 61
ROOT_DIR = "non-incremental"
RESULTS_DIR = "results_cvc4_cegqi"
```

Listing 6.5: Configuration for CEGQI

The approach of [Niemetz et al., 2021] was implemented in CVC4, can be found here: `https://github.com/pysmt/CVC4` and is tested with the following configuration:

```
SOLVER = "cvc4"
SOLVER_OPTS = ["--sygus"]
TIMEOUT = 61
ROOT_DIR = "non-incremental"
RESULTS_DIR = "results_cvc4_syqic"
```

Listing 6.6: Configuration for SyQI

The approach of [Schoisswohl et al., 2024] is not implemented to date, and therefore, benchmarking is not possible.

The approach of [Garcia-Contreras et al., 2023] was implemented in Z3, can be found here: `https://github.com/Z3Prover/z3` and is tested with the following configuration:

```
SOLVER = "z3"
SOLVER_OPTS = []
TIMEOUT = 61
ROOT_DIR = "non-incremental"
RESULTS_DIR = "results_QEL"
```

Listing 6.7: Configuration for Z3 QEL

The approach of [Bjørner and Janota, 2015] was implemented in Z3, can be found here: `https://github.com/Z3Prover/z3` and is tested with the following configuration:

```
SOLVER = "z3"
SOLVER_OPTS = []
TIMEOUT = 61
ROOT_DIR = "non-incremental"
RESULTS_DIR = "results_QSAT"
```

Listing 6.8: Configuration for Z3 QSAT

6.3 Results

Note that each approach has its own strengths, and some methods are more tailored to LRA and others to LIA. Take also into account that no benchmarks for LIRA were performed.

The following bar charts show the results of the benchmarks; each benchmark was run with 61 seconds timeout. For simplicity and because this is not the main topic of the thesis, the results were interpreted by checking only how many instances they solved and how often they timed out. We also generated tables, such that we can see how each of them performed in the LRA and LIA categories.

ALASCA achieved strong results on quantified LRA benchmarks, and as it was implemented in Vampire, we can clearly see the strengths of the theorem prover. Beagle, on the other hand, performed as expected, as it is a rather old approach. Instantiation-based methods, such as CEGQI and SYQI, consistently performed well on all sorts of problems, but that is expected as CVC4 and further CVC5 are among the strongest SMT-solvers. The instantiation approach implemented in iProver did not perform as well as, for example ALASCA in Vampire. Quantifier elimination techniques, such as QEL and QSAT, implemented in Z3 performed best overall, as they only timed out for 12, respectively 11 problems.

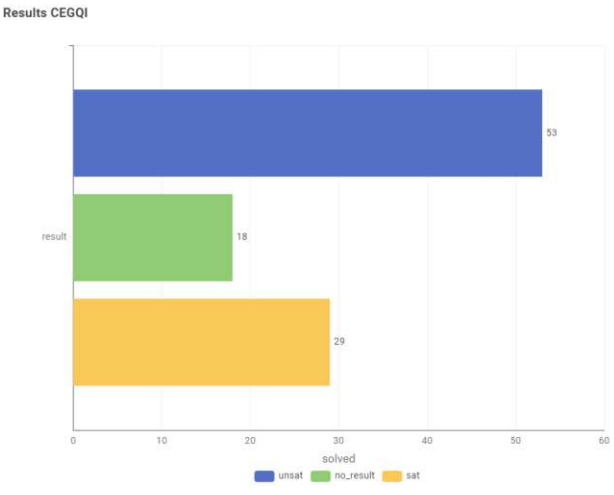

Figure 6.1: Results for CEGQI

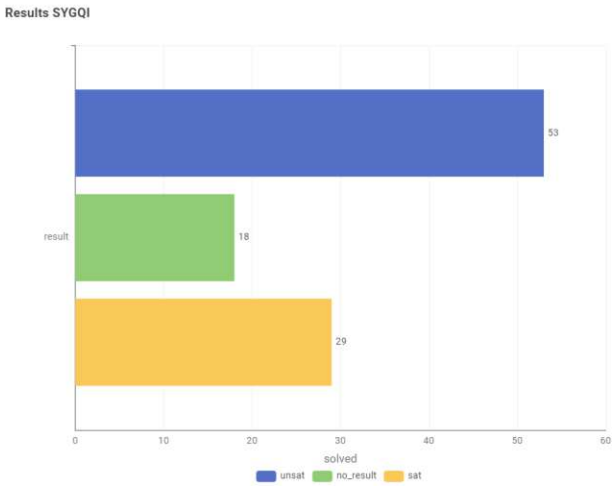

Figure 6.2: Results for SYQI

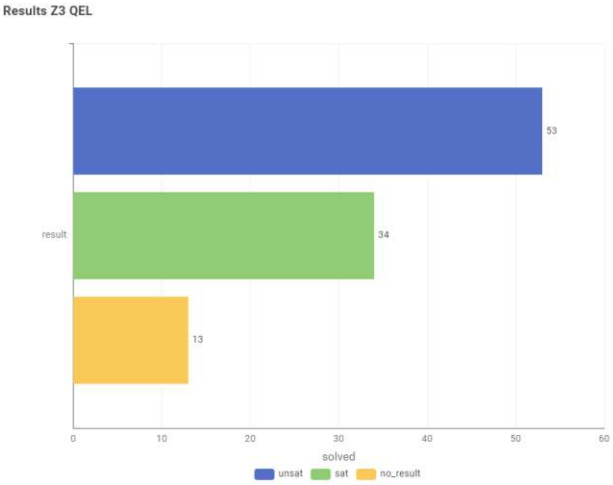

Figure 6.3: Results for Z3 QEL

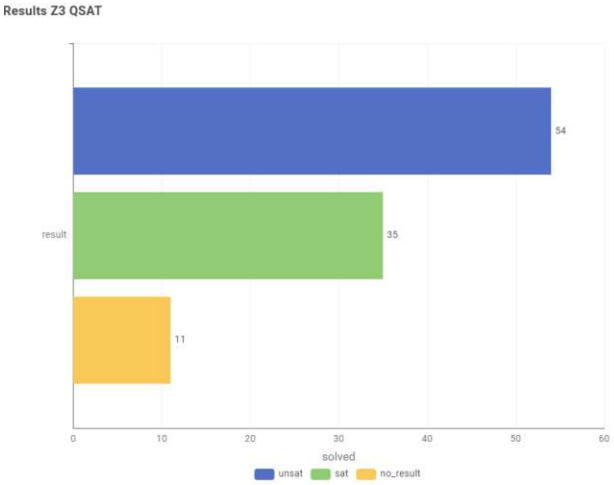

Figure 6.4: Results for Z3 QSAT

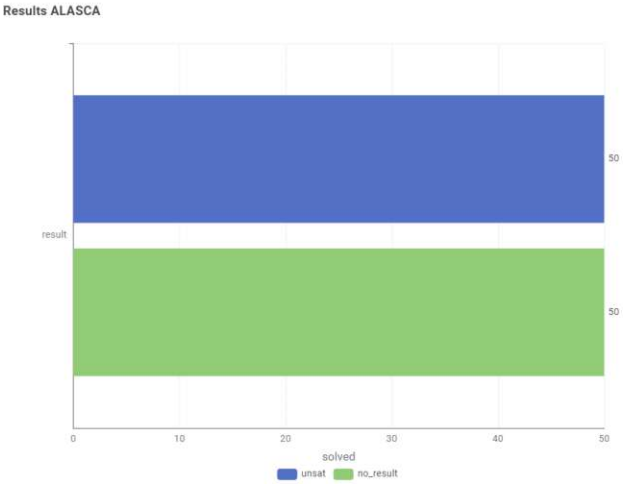

Figure 6.5: Results for ALASCA

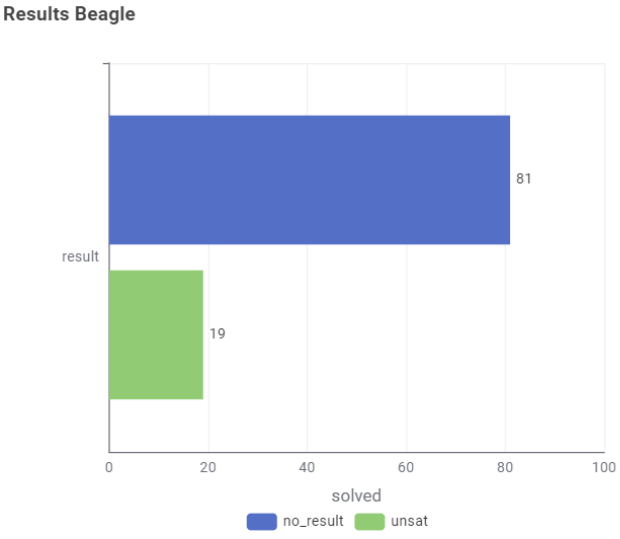

Figure 6.6: Results for Beagle

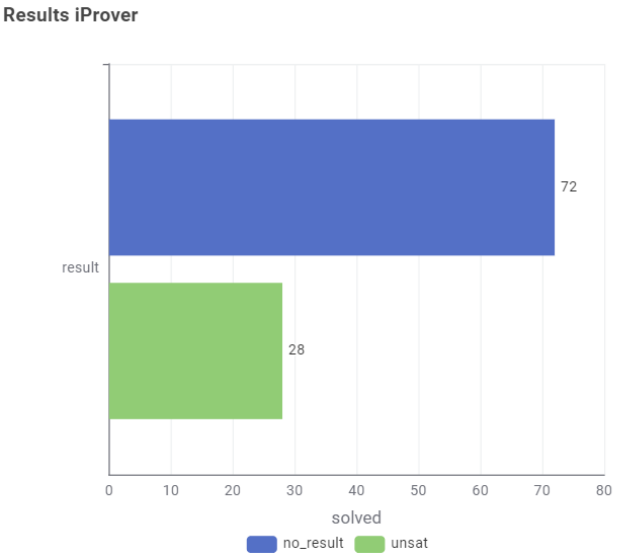

Figure 6.7: Results for iProver

	LRA	LIA
sat	23	6
unsat	40	13
no_result	14	4

Table 6.1: Results for CEGQI

	LRA	LIA
sat	23	6
unsat	40	13
no_result	14	4

Table 6.2: Results for SYQI

	LRA	LIA
sat	27	7
unsat	43	10
no_result	7	6

Table 6.3: Results for Z3 QEL

	LRA	LIA
sat	28	7
unsat	44	10
no_result	5	6

Table 6.4: Results for Z3 QSAT

	LRA	LIA
sat	nan	nan
unsat	41	9
no_result	36	14

Table 6.5: Results for ALASCA

	LRA	LIA
sat	nan	nan
unsat	7	12
no_result	70	11

Table 6.6: Results for BEAGLE

	LRA	LIA
sat	nan	nan
unsat	16	12
no_result	61	11

Table 6.7: Results for iProver

CHAPTER 7

Conclusion

This thesis addressed the challenge of automated reasoning for quantified formulas over LA, which is a problem central to verification, synthesis and constraint solving due to the expressive power of quantifiers over linear constraints. We have summarized various approaches for three categories of quantified reasoning in linear arithmetic. The quantifier-free fragment of LA is well understood and is efficiently addressed, however, the introduction of quantifiers remains a significant challenge. While handling quantifiers in LA is a significant challenge, it is also very important, as it enables us to address much deeper questions and solve more general problems that are necessary for practical use cases. Most of the problems involve LRA and LIA. Both theories are decidable, but quantifier elimination methods are often costly due to their worst-case complexity, which is double- or even in some cases triple-exponential. And on top, in real world applications, quantified formulas mostly involve also uninterpreted functions, arrays or non-linear constructs, which even enhance the complexity by a lot.

After giving an introduction to automated reasoning and its applications, and also LA and its reasoning methods, we analyzed and surveyed various approaches in three categories: superposition-based methods, instantiation-based methods and quantifier elimination methods in computer algebra. In each category, we looked at three methods. For superposition-based methods they were: [Korovin et al., 2023], [Baumgartner et al., 2015], [Althaus et al., 2009]. For instantiation-based methods: [Reynolds et al., 2017], [Ganzinger and Korovin, 2006], [Niemetz et al., 2021] and for quantifier elimination methods in computer algebra: [Schoisswohl et al., 2024], [Garcia-Contreras et al., 2023], [Bjørner and Janota, 2015]. The comparison included a complexity analysis, soundness and completeness results, practical use cases, experimental results, as well as limitations and future work. In addition, some benchmarks were performed as well as possible, but note that these were not the main contribution of the thesis.

The study of the introduced approaches demonstrates that each family of methods

has distinct strengths, and each one has individual limitations. Our comparison shows that superposition-based lifting, in particular ALASCA, performs very well on many LRA problems with uninterpreted functions and yields strong solver performance in practice. Its ability to avoid instantiation as often as possible while preserving completeness allowed it to outperform related calculi such as SUP(LA) and Beagle. Instantiation-based methods, on the other hand, showed their greatest advantage in fragments with limited quantifier alternation. Counterexample-guided instantiation demonstrated strong performance, producing instances without the combinatorial explosion that other approaches exhibit. Syntax-guided instantiation performed really well on arithmetic fragments where syntactic patterns could be exploited. Overall, these approaches offer a good balance between theoretical limits and solver performance in practice, therefore, they are really suitable for applications in program verification and synthesis. Classic quantifier-elimination techniques such as Fourier–Motzkin are complete but rarely scale and are effective in practice. Model-based projection and conflict-driven elimination showed that controlled incompleteness can in fact be a practical advantage. By trading completeness for scalability, these approaches solve many instances quickly and offer efficient ways to integrate algebraic reasoning into SMT workflows. All in all, it is evident that no single approach offers a universal solution. Each method itself can be viewed as optimized for a particular application.

As a result, we have no universal winner, each approach dominates in a different category of problems and also in a different way. That suggests that a single approach cannot solve quantified reasoning in LA. Instead, we could see hybrid architectures that combine all approaches. In recent years, solvers have evolved a lot, and nowadays, they can already solve many non-trivial quantified benchmarks that were out of reach a decade ago. However, note that truly deep alternations and mixed-theory problems are still one of the most challenging open challenges.

As the field of artificial intelligence has evolved significantly in recent years, this progress is also promising for future modern SMT-solvers and theorem provers. Nevertheless, this thesis has shown that while significant progress has been made in reasoning about quantified formulas in LA, the landscape remains fragmented across different methods. The absence of a universal method is not a weakness, but rather an indication that quantified reasoning is a topic with numerous facets, as well as numerous possibilities, and is not trivial by any means. This requires solvers to adapt their strategies to the structure of the input problem. The observation of this thesis implies that the most promising direction lies not in hunting a single perfect calculus, but rather in designing hybrid architectures that can dynamically combine all possible approaches. Intelligently doing that is a big topic for future research. Such systems would not only address the current limitations of scalability and some completeness results, but also expand the applicability of automated reasoning to verification, synthesis, and constraint-solving tasks of increasing complexity, and many more.

Overview of Generative AI Tools Used

In this thesis, generative AI tools were utilized to support both the writing and coding aspects of the work. Specifically, ChatGPT was employed to assist with formulating clear and coherent text, helping to refine explanations and improve overall readability. For programming tasks, GitHub Copilot was used to generate and suggest code snippets.

List of Figures

List of Tables

Bibliography

[Alt et al., 2022] Alt, L., Blicha, M., Hyvärinen, A. E., and Sharygina, N. (2022). Solcmc: Solidity compiler's model checker. In *International Conference on Computer Aided Verification*, pages 325–338. Springer.

[Althaus et al., 2009] Althaus, E., Kruglov, E., and Weidenbach, C. (2009). Superposition modulo linear arithmetic sup(la). In Ghilardi, S. and Sebastiani, R., editors, *Frontiers of Combining Systems*, pages 84–99, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Andreotti et al., 2022] Andreotti, B., Barbosa, H., Fontaine, P., and Schurr, H. (2022). verit at smt-comp 2022. `https://smt-comp.github.io/2022/system-descriptions/veriT.pdf`.

[Armando and Compagna, 2004] Armando, A. and Compagna, L. (2004). Satmc: A sat-based model checker for security protocols. In Alferes, J. J. and Leite, J., editors, *Logics in Artificial Intelligence*, pages 730–733, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Arnon et al., 1984] Arnon, D. S., Collins, G. E., and McCallum, S. (1984). Cylindrical algebraic decomposition i: The basic algorithm. *SIAM Journal on Computing*, 13(4):865–877.

[Baader and Nipkow, 1998] Baader, F. and Nipkow, T. (1998). *Term rewriting and all that*. Cambridge university press.

[Bachmair and Ganzinger, 1994] Bachmair, L. and Ganzinger, H. (1994). Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247.

[Bachmair et al., 2001] Bachmair, L., Ganzinger, H., McAllester, D., and Lynch, C. (2001). Chapter 2 - resolution theorem proving. In Robinson, A. and Voronkov, A., editors, *Handbook of Automated Reasoning*, Handbook of Automated Reasoning, pages 19–99. North-Holland, Amsterdam.

[Bachmair et al., 1994] Bachmair, L., Ganzinger, H., and Waldmann, U. (1994). Refutational theorem proving for hierarchic first-order theories. *Applicable Algebra in Engineering, Communication and Computing*, 5(3):193–212.

[Barbosa et al., 2022] Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., et al. (2022). Cvc5 at the smt competition 2022. https://smt-comp.github.io/2022/system-descriptions/cvc5.pdf.

[Barendregt et al., 2013] Barendregt, H. P., Dekkers, W., and Statman, R. (2013). *Lambda calculus with types*. Cambridge University Press.

[Barrett et al., 2011] Barrett, C., Conway, C. L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., and Tinelli, C. (2011). Cvc4. In Gopalakrishnan, G. and Qadeer, S., editors, *Computer Aided Verification*, pages 171–177, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Barrett et al., 2005] Barrett, C., de Moura, L., and Stump, A. (2005). Design and results of the first satisfiability modulo theories competition (smt-comp 2005). *Journal of Automated Reasoning*, 35(4):373–390.

[Barrett et al., 2016] Barrett, C., Fontaine, P., and Tinelli, C. (2016). The satisfiability modulo theories library (smt-lib). www. *SMT-LIB. org*, 2:68.

[Bártek et al., 2025] Bártek, F., Bhayat, A., Coutelier, R., Hajdu, M., Hetzenberger, M., Hozzová, P., Kovács, L., Rath, J., Rawson, M., Reger, G., Suda, M., Schoisswohl, J., and Voronkov, A. (2025). The vampire diary. In Piskac, R. and Rakamarić, Z., editors, *Computer Aided Verification*, pages 57–71, Cham. Springer Nature Switzerland.

[Barth et al., 2022] Barth, M., Dietsch, D., Heizmann, M., and Podelski, A. (2022). Ultimate eliminator at smt-comp 2022. https://smt-comp.github.io/2022/system-descriptions/UltimateEliminator%2BMathSAT.pdf.

[Baumgartner et al., 2015] Baumgartner, P., Bax, J., and Waldmann, U. (2015). Beagle – a hierarchic superposition theorem prover. In Felty, A. P. and Middeldorp, A., editors, *Automated Deduction - CADE-25*, pages 367–377, Cham. Springer International Publishing.

[Baumgartner et al., 1996] Baumgartner, P., Furbach, U., and Niemelä, I. (1996). Hyper tableaux. In *European Workshop on Logics in Artificial Intelligence*, pages 1–17. Springer.

[Baumgartner and Tinelli, 2003] Baumgartner, P. and Tinelli, C. (2003). The model evolution calculus. volume 2741, pages 350–364.

[Bergsträßer et al., 2024] Bergsträßer, P., Ganardi, M., Lin, A. W., and Zetzsche, G. (2024). Ramsey quantifiers in linear arithmetics. *Proc. ACM Program. Lang.*, 8(POPL).

[Bibel, 1983] Bibel, W. (1983). Matings in matrices. *Communications of the ACM*, 26(11):844–852.

[Biere et al., 2009] Biere, A., Heule, M., and van Maaren, H. (2009). *Handbook of satisfiability*, volume 185. IOS press.

[Bjørner and Janota, 2015] Bjørner, N. S. and Janota, M. (2015). Playing with quantified satisfaction. In Fehnker, A., McIver, A., Sutcliffe, G., and Voronkov, A., editors, *20th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning - Short Presentations, LPAR 2015, Suva, Fiji, November 24-28, 2015*, volume 35 of *EPiC Series in Computing*, pages 15–27. EasyChair.

[Boigelot et al., 2001] Boigelot, B., Jodogne, S., and Wolper, P. (2001). On the use of weak automata for deciding linear arithmetic with integer and real variables. In Goré, R., Leitsch, A., and Nipkow, T., editors, *Automated Reasoning*, pages 611–625, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Boigelot et al., 2005] Boigelot, B., Jodogne, S., and Wolper, P. (2005). An effective decision procedure for linear arithmetic over the integers and reals. *ACM Trans. Comput. Logic*, 6(3):614–633.

[Boigelot et al., 1998] Boigelot, B., Rassart, S., and Wolper, P. (1998). On the expressiveness of real and integer arithmetic automata. In Larsen, K. G., Skyum, S., and Winskel, G., editors, *Automata, Languages and Programming*, pages 152–163, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Boigelot and Wolper, 2002] Boigelot, B. and Wolper, P. (2002). Representing arithmetic constraints with finite automata: An overview. In Stuckey, P. J., editor, *Logic Programming*, pages 1–20, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Bonacina, 2001] Bonacina, M. P. (2001). A taxonomy of theorem-proving strategies. In *Artificial Intelligence Today: Recent Trends and Developments*, pages 43–84. Springer.

[Bonacina et al., 2017] Bonacina, M. P., Graham-Lengrand, S., and Shankar, N. (2017). Satisfiability modulo theories and assignments. In de Moura, L., editor, *Automated Deduction – CADE 26*, pages 42–59, Cham. Springer International Publishing.

[Bonacina et al., 2023] Bonacina, M. P., Graham-Lengrand, S., and Vauthier, C. (2023). Qsma: A new algorithm for quantified satisfiability modulo theory and assignment. In Pientka, B. and Tinelli, C., editors, *Automated Deduction – CADE 29*, pages 78–95, Cham. Springer Nature Switzerland.

[Bordeaux and Zhang, 2007] Bordeaux, L. and Zhang, L. (2007). A solver for quantified boolean and linear constraints. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 321–325.

[Bromberger et al., 2020] Bromberger, M., Sturm, T., and Weidenbach, C. (2020). A complete and terminating approach to linear integer solving. *Journal of Symbolic Computation*, 100:102–136.

[Brown, 2003] Brown, C. W. (2003). Qepcad b: a program for computing with semi-algebraic sets using cads. *SIGSAM Bull.*, 37(4):97–108.

[Buresh-Oppenheim and Pitassi, 2007] Buresh-Oppenheim, J. and Pitassi, T. (2007). The complexity of resolution refinements. *The Journal of Symbolic Logic*, 72(4):1336–1352.

[Chatterjee et al., 2025] Chatterjee, K., Goharshady, E. K., Karrabi, M., Motwani, H. J., Seeliger, M., and Žikelić, D. o. e. (2025). Quantified linear and polynomial arithmetic satisfiability via template-based skolemization. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(11):11158–11166.

[Collins, 1976] Collins, G. E. (1976). Quantifier elimination for real closed fields by cylindrical algebraic decomposition: A synopsis. *SIGSAM Bull.*, 10(1):10–12.

[Cook, 2018] Cook, B. (2018). Formal reasoning about the security of amazon web services. In Chockler, H. and Weissenbacher, G., editors, *Computer Aided Verification*, pages 38–47, Cham. Springer International Publishing.

[Cook, 1971] Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158, New York, NY, USA. Association for Computing Machinery.

[Cooper, 1972] Cooper, D. C. (1972). Theorem proving in arithmetic without multiplication. *Machine intelligence*, 7(91-99):300.

[Cruanes, 2015] Cruanes, S. (2015). *Extending Superposition with Integer Arithmetic, Structural Induction, and Beyond*. Theses, École polytechnique.

[Davis and Putnam, 1960] Davis, M. and Putnam, H. (1960). A computing procedure for quantification theory. *J. ACM*, 7(3):201–215.

[de Moura and Bjørner, 2007] de Moura, L. and Bjørner, N. (2007). Efficient e-matching for smt solvers. In Pfenning, F., editor, *Automated Deduction – CADE-21*, pages 183–198, Berlin, Heidelberg. Springer Berlin Heidelberg.

[de Moura and Bjørner, 2009] de Moura, L. and Bjørner, N. (2009). Satisfiability modulo theories: An appetizer. In Oliveira, M. V. M. and Woodcock, J., editors, *Formal Methods: Foundations and Applications*, pages 23–36, Berlin, Heidelberg. Springer Berlin Heidelberg.

[De Moura and Jovanović, 2013] De Moura, L. and Jovanović, D. (2013). A model-constructing satisfiability calculus. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 1–12. Springer.

[Desharnais et al., 2022] Desharnais, M., Vukmirović, P., Blanchette, J., and Wenzel, M. (2022). Seventeen provers under the hammer. In *13th International Conference on Interactive Theorem Proving-ITP 2022*.

[Distefano et al., 2019] Distefano, D., Fähndrich, M., Logozzo, F., and O'Hearn, P. W. (2019). Scaling static analyses at facebook. *Commun. ACM*, 62(8):62–70.

[Dolzmann et al., 1999] Dolzmann, A., Sturm, T., and Weispfenning, V. (1999). *Real Quantifier Elimination in Practice*.

[Dutertre and de Moura, 2006] Dutertre, B. and de Moura, L. (2006). A fast linear-arithmetic solver for dpll(t). In Ball, T. and Jones, R. B., editors, *Computer Aided Verification*, pages 81–94, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Ehlers, 2017] Ehlers, R. (2017). Formal verification of piece-wise linear feed-forward neural networks.

[Elad et al., 2021] Elad, N., Rain, S., Immerman, N., Kovács, L., and Sagiv, M. (2021). Summing up smart transitions. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I 33*, pages 317–340. Springer.

[Farkas, 1902] Farkas, J. (1902). Theorie der einfachen ungleichungen. *Journal für die reine und angewandte Mathematik (Crelles Journal)*, 1902(124):1–27.

[Ferrante and Rackoff, 1975] Ferrante, J. and Rackoff, C. (1975). A decision procedure for the first order theory of real addition with order. *SIAM Journal on Computing*, 4(1):69–76.

[Ferrante and Rackoff, 2006] Ferrante, J. and Rackoff, C. W. (2006). *The computational complexity of logical theories*, volume 718. Springer.

[Fourier, 1827] Fourier, J. (1827). Histoire de l'académie, partie mathématique (1824). *Mémoires de l'Académie des sciences de l'Institut de France*, 7:38.

[Ganzinger et al., 2003] Ganzinger, H., Hillenbrand, T., and Waldmann, U. (2003). Superposition modulo a shostak theory. In *Automated Deduction–CADE-19: 19th International Conference on Automated Deduction, Miami Beach, FL, USA, July 28–August 2, 2003. Proceedings 19*, pages 182–196. Springer.

[Ganzinger and Korovin, 2003] Ganzinger, H. and Korovin, K. (2003). New directions in instantiation-based theorem proving. In *18th Annual IEEE Symposium of Logic in Computer Science, 2003. Proceedings.*, pages 55–64. IEEE.

[Ganzinger and Korovin, 2004] Ganzinger, H. and Korovin, K. (2004). Integrating equational reasoning into instantiation-based theorem proving. In *Computer Science Logic: 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Karpacz, Poland, September 20-24, 2004. Proceedings 18*, pages 71–84. Springer.

[Ganzinger and Korovin, 2006] Ganzinger, H. and Korovin, K. (2006). Theory instantiation. In Hermann, M. and Voronkov, A., editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 497–511, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Garcia-Contreras et al., 2023] Garcia-Contreras, I., K, H. G. V., Shoham, S., and Gurfinkel, A. (2023). Fast approximations of quantifier elimination.

[Ge and de Moura, 2009] Ge, Y. and de Moura, L. (2009). Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In Bouajjani, A. and Maler, O., editors, *Computer Aided Verification*, pages 306–320, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Gentzen, 1935] Gentzen, G. (1935). Untersuchungen über das logische schließen. i. *Mathematische zeitschrift*, 39(1):176–210.

[Giese, 2001] Giese, M. (2001). Incremental closure of free variable tableaux. In Goré, R., Leitsch, A., and Nipkow, T., editors, *Automated Reasoning*, pages 545–560, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Godoy and Nieuwenhuis, 2004] Godoy, G. and Nieuwenhuis, R. (2004). Superposition with completely built-in abelian groups. *Journal of Symbolic Computation*, 37(1):1–33.

[Goldberg and Manolios, 2014] Goldberg, E. and Manolios, P. (2014). Quantifier elimination by dependency sequents. *Formal Methods in System Design*, 45:111–143.

[Goultiaeva et al., 2013] Goultiaeva, A., Seidl, M., and Biere, A. (2013). Bridging the gap between dual propagation and cnf-based qbf solving. In *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 811–814. IEEE.

[Graf and Fehrer, 1998] Graf, P. and Fehrer, D. (1998). *Term Indexing*, pages 125–147. Springer Netherlands, Dordrecht.

[Graham-Lengrand, 2021] Graham-Lengrand, S. (2021). Yices-qs, an extension of yices for quantified satisfiability.

[Graham-Lengrand, 2022] Graham-Lengrand, S. (2022). Yices-qs 2022, an extension of yices for quantified satisfiability. `https://smt-comp.github.io/2022/system-descriptions/YicesQS.pdf`.

[Grumberg et al., 1999] Grumberg, O., Clarke, E., and Peled, D. (1999). Model checking. In *International Conference on Foundations of Software Technology and Theoretical Computer Science; Springer: Berlin/Heidelberg, Germany.*

[Gurfinkel, 2022] Gurfinkel, A. (2022). Program verification with constrained horn clauses. In *International Conference on Computer Aided Verification*, pages 19–29. Springer.

[Gurfinkel et al., 2018] Gurfinkel, A., Ruemmer, P., Fedyukovich, G., and Champion, A. (2018). Chc-comp. `https://chc-comp.github.io/`.

[Habermehl et al., 2024] Habermehl, P., Havlena, V., Hečko, M., Holík, L., and Lengál, O. (2024). Algebraic reasoning meets automata in solving linear integer arithmetic. In *Computer Aided Verification: 36th International Conference, CAV 2024, Montreal,*

QC, Canada, July 24–27, 2024, Proceedings, Part I, page 42–67, Berlin, Heidelberg. Springer-Verlag.

[Herbrand, 1930] Herbrand, J. (1930). *Recherches sur la théorie de la démonstration*.

[Hintikka, 1982] Hintikka, J. (1982). Game-theoretical semantics: Insights and prospects. *Notre Dame Journal of Formal Logic*, 23(2):219–241.

[Hoenicke and Schindler, 2022] Hoenicke, J. and Schindler, T. (2022). Smtinterpol with resolution proofs. `https://smt-comp.github.io/2022/system-descriptions/smtinterpol.pdf`.

[Hojjat and Rümmer, 2018] Hojjat, H. and Rümmer, P. (2018). The eldarica horn solver. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–7. IEEE.

[Huth and Ryan, 2004] Huth, M. and Ryan, M. (2004). *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge university press.

[Jakubův and Janota, 2025] Jakubův, J. and Janota, M. (2025). Quantifier instantiations: To mimic or to revolt?

[Janota et al., 2016] Janota, M., Klieber, W., Marques-Silva, J., and Clarke, E. (2016). Solving qbf with counterexample guided refinement. *Artificial Intelligence*, 234:1–25.

[Janota and Marques-Silva, 2015] Janota, M. and Marques-Silva, J. (2015). Solving qbf by clause selection. In *IJCAI*, pages 325–331.

[Jha et al., 2009] Jha, S., Limaye, R., and Seshia, S. A. (2009). Beaver: Engineering an efficient smt solver for bit-vector arithmetic. In Bouajjani, A. and Maler, O., editors, *Computer Aided Verification*, pages 668–674, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Jirstrand, 1995] Jirstrand, M. (1995). *Cylindrical algebraic decomposition-an introduction*. Linköping University.

[Jovanović and de Moura, 2013] Jovanović, D. and de Moura, L. (2013). Solving nonlinear arithmetic. *ACM Commun. Comput. Algebra*, 46(3/4):104–105.

[Kapur and Narendran, 1992] Kapur, D. and Narendran, P. (1992). Double-exponential complexity of computing a complete set of ac-unifiers. In *[1992] Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 11–21.

[Knuth and Bendix, 1983] Knuth, D. E. and Bendix, P. B. (1983). Simple word problems in universal algebras. *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*, pages 342–376.

[Komuravelli et al., 2014] Komuravelli, A., Gurfinkel, A., and Chaki, S. (2014). Smtbased model checking for recursive programs.

[Korovin, 2008] Korovin, K. (2008). iprover – an instantiation-based theorem prover for first-order logic (system description). In Armando, A., Baumgartner, P., and Dowek, G., editors, *Automated Reasoning*, pages 292–298, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Korovin et al., 2023] Korovin, K., Kovács, L., Reger, G., Schoisswohl, J., and Voronkov, A. (2023). Alasca: Reasoning in quantified linear arithmetic. In Sankaranarayanan, S. and Sharygina, N., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 647–665, Cham. Springer Nature Switzerland.

[Korovin and Voronkov, 2007] Korovin, K. and Voronkov, A. (2007). Integrating linear arithmetic into superposition calculus. In Duparc, J. and Henzinger, T. A., editors, *Computer Science Logic*, pages 223–237, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Kovács and Voronkov, 2013] Kovács, L. and Voronkov, A. (2013). First-order theorem proving and vampire. In Sharygina, N. and Veith, H., editors, *Computer Aided Verification*, pages 1–35, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Kripke, 1963] Kripke, S. (1963). Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16:83–94.

[Letz and Stenz, 2002] Letz, R. and Stenz, G. (2002). Integration of equality reasoning into the disconnection calculus. In Egly, U. and Fermüller, C. G., editors, *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 176–190, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Lintao et al., 2001] Lintao, Z., Madigan, C., Moskewicz, M., and Malik, S. (2001). Efficient conflict driven learning in a boolean satisfiability solver. pages 279–285.

[Lloyd, 2012] Lloyd, J. W. (2012). *Foundations of logic programming*. Springer Science & Business Media.

[Loos and Weispfenning, 1993] Loos, R. and Weispfenning, V. (1993). Applying linear quantifier elimination. *The Computer Journal*, 36(5):450–462.

[Loveland, 1970] Loveland, D. W. (1970). A linear format for resolution. In Laudet, M., Lacombe, D., Nolin, L., and Schützenberger, M., editors, *Symposium on Automatic Demonstration*, pages 147–162, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Matoušek and Gärtner, 2007] Matoušek, J. and Gärtner, B. (2007). *Understanding and Using Linear Programming*.

[Monniaux, 2008] Monniaux, D. (2008). A quantifier elimination algorithm for linear real arithmetic. In Cervesato, I., Veith, H., and Voronkov, A., editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 243–257, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Motzkin, 1936] Motzkin, T. S. (1936). Beiträge zur theorie der linearen ungleichungen. *(No Title)*.

[Murphy and Kincaid, 2024] Murphy, C. and Kincaid, Z. (2024). Quantified linear arithmetic satisfiability via fine-grained strategy improvement. In Gurfinkel, A. and Ganesh, V., editors, *Computer Aided Verification*, pages 89–109, Cham. Springer Nature Switzerland.

[Nadel and Ryvchin, 2018] Nadel, A. and Ryvchin, V. (2018). *Chronological Backtracking*, pages 111–121.

[Nalbach et al., 2023] Nalbach, J., Promies, V., Ábrahám, E., and Kobialka, P. (2023). Fmplex: A novel method for solving linear real arithmetic problems.

[Nehai and Bobot, 2019] Nehai, Z. and Bobot, F. (2019). Deductive proof of ethereum smart contracts using why3.

[Nelson and Oppen, 1979] Nelson, G. and Oppen, D. C. (1979). Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):245–257.

[Niemetz et al., 2021] Niemetz, A., Preiner, M., Reynolds, A., Barrett, C., and Tinelli, C. (2021). Syntax-guided quantifier instantiation. In Groote, J. F. and Larsen, K. G., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 145–163, Cham. Springer International Publishing.

[Passmore, 2021] Passmore, G. O. (2021). Some lessons learned in the industrialization of formal methods for financial algorithms. In *Formal Methods: 24th International Symposium, FM 2021, Virtual Event, November 20–26, 2021, Proceedings 24*, pages 717–721. Springer.

[Pearl et al., 2000] Pearl, J. et al. (2000). Models, reasoning and inference. *Cambridge, UK: CambridgeUniversityPress*, 19(2):3.

[Phan et al., 2012] Phan, A.-D., Bjørner, N., and Monniaux, D. (2012). Anatomy of alternating quantifier satisfiability (work in progress). In *10th International Workshop on Satisfiability Modulo Theories*, page 6.

[Portoraro, 2001] Portoraro, F. (2001). Automated reasoning.

[Prawitz, 2006] Prawitz, D. (2006). *Natural deduction: A proof-theoretical study*. Courier Dover Publications.

[Preiner et al., 2025] Preiner, M., Schurr, H.-J., Barrett, C., Fontaine, P., Niemetz, A., and Tinelli, C. (2025). Smt-lib release 2025 (non-incremental benchmarks).

[Presburger, 1929] Presburger, M. (1929). Uber die vollstandigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchen die addition als einzige operation hervortritt. In *Comptes-Rendus du ler Congres des Mathematiciens des Pays Slavs*.

[Pugh, 1991] Pugh, W. (1991). The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, page 4–13, New York, NY, USA. Association for Computing Machinery.

[Pugh, 1992] Pugh, W. (1992). A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114.

[Reddy and Loveland, 1978] Reddy, C. R. and Loveland, D. W. (1978). Presburger arithmetic with bounded quantifier alternation. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 320–325.

[Reger et al., 2018] Reger, G., Suda, M., and Voronkov, A. (2018). Unification with abstraction and theory instantiation in saturation-based reasoning. In Beyer, D. and Huisman, M., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 3–22, Cham. Springer International Publishing.

[Reger et al., 2022] Reger, G., Suda, M., Voronkov, A., Kovács, L., Bhayat, A., Gleiss, B., Hajdu, M., Hozzova, P., Evgeny Kotelnikov, J., Rawson, M., et al. (2022). Vampire 4.7-smt system description. `https://smt-comp.github.io/2022/system-descriptions/Vampire.pdf`.

[Reynolds et al., 2017] Reynolds, A., King, T., and Kuncak, V. (2017). Solving quantified linear arithmetic by counterexample-guided instantiation. *Formal Methods in System Design*, 51(3):500–532.

[Robinson and Voronkov, 2001] Robinson, A. J. and Voronkov, A. (2001). *Handbook of automated reasoning*, volume 1. Elsevier.

[Robinson, 1965a] Robinson, J. (1965a). Automatic deduction with hyper-resolution. *Int. J. Comput. Math.*, 1:227–234.

[Robinson, 1965b] Robinson, J. A. (1965b). A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41.

[Rümmer, 2008] Rümmer, P. (2008). A constraint sequent calculus for first-order logic with linear integer arithmetic. In Cervesato, I., Veith, H., and Voronkov, A., editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 274–289, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Schoisswohl et al., 2024] Schoisswohl, J., Kovács, L., and Korovin, K. (2024). Viras: Conflict-driven quantifier elimination for integer-real arithmetic. In *# PLACEHOLDER_PARENT_METADATA_VALUE#*, volume 100, pages 147–164.

[Schulz et al., 2019] Schulz, S., Cruanes, S., and Vukmirović, P. (2019). Faster, higher, stronger: E 2.3. In *Automated Deduction–CADE 27: 27th International Conference on Automated Deduction, Natal, Brazil, August 27–30, 2019, Proceedings 27*, pages 495–507. Springer.

[Sidrane et al., 2021] Sidrane, C., Maleki, A., Irfan, A., and Kochenderfer, M. J. (2021). Overt: An algorithm for safety verification of neural network control policies for nonlinear systems.

[SMT-LIB,] SMT-LIB. The satisfiability modulo theories librar. `http://smt-lib.org/`.

[So et al., 2019] So, S., Lee, M., Park, J., Lee, H., and Oh, H. (2019). Verismart: A highly precise safety verifier for ethereum smart contracts.

[Stansifer, 1984] Stansifer, R. (1984). Presburger's article on integer arithmetic: Remarks and translation. Technical report, Cornell University.

[Stuber, 1998] Stuber, J. (1998). Superposition theorem proving for abelian groups represented as integer modules. *Theor. Comput. Sci.*, 208(1-2):149–177.

[Sturm, 2017] Sturm, T. (2017). A survey of some methods for real quantifier elimination, decision, and satisfiability and their applications. *Mathematics in Computer Science*, 11(3):483–502.

[Sutcliffe, 2009] Sutcliffe, G. (2009). The tptp problem library and associated infrastructure. *Journal of Automated Reasoning*, 43(4):337–362.

[Sutcliffe, 2015] Sutcliffe, G. (2015). The 7th ijcar automated theorem proving system competition–casc-j7. *Ai communications*, 28(4):683–692.

[Urquhart, 1987] Urquhart, A. (1987). Hard examples for resolution. *J. ACM*, 34(1):209–219.

[Voronkov, 2014] Voronkov, A. (2014). Avatar: The architecture for first-order theorem provers. In Biere, A. and Bloem, R., editors, *Computer Aided Verification*, pages 696–710, Cham. Springer International Publishing.

[Waldmann, 1998] Waldmann, U. (1998). Superposition for divisible torsion-free abelian groups. In Kirchner, C. and Kirchner, H., editors, *Automated Deduction — CADE-15*, pages 144–159, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Waldmann, 2001] Waldmann, U. (2001). Superposition and chaining for totally ordered divisible abelian groups. In Goré, R., Leitsch, A., and Nipkow, T., editors, *Automated Reasoning*, pages 226–241, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Waldmann, 2002] Waldmann, U. (2002). Cancellative abelian monoids and related structures in refutational theorem proving (part i,ii). *Journal of Symbolic Computation*, 33:777–829, 831–861.

[Weispfenning, 1997] Weispfenning, V. (1997). Quantifier elimination for real algebra—the quadratic case and beyond. *Applicable Algebra in Engineering, Communication and Computing*, 8:85–101.

[Willsey et al., 2021] Willsey, M., Nandi, C., Wang, Y. R., Flatt, O., Tatlock, Z., and Panchekha, P. (2021). egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL).

[Wos et al., 1986] Wos, L., Overbeek, R., Lusk, E., and Boyle, J. (1986). Automated reasoning. introduction and applications. *Journal of Symbolic Logic*, 51(2):464–465.

[Wos et al., 1965] Wos, L., Robinson, G. A., and Carson, D. F. (1965). Efficiency and completeness of the set of support strategy in theorem proving. *J. ACM*, 12(4):536–541.

[Zhang, 2006] Zhang, L. (2006). Solving qbf with combined conjunctive and disjunctive normal form. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, volume 21, page 143. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999.